UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Detección automática de problemas de accesibilidad a partir de eventos de interacción de usuario

AUTORES: Maximiliano Jonathan Toledo

DIRECTOR: Dra. Alejandra Garrido

CODIRECTOR: Dr. Julián Grigera

ASESOR PROFESIONAL: ----

CARRERA: Licenciatura en Sistemas

Resumen

Actualmente muchas de las actividades de nuestra vida cotidiana se encuentran integradas en una aplicación web. Toda la información de los hechos que acontecen en el mundo está a un simple clic, por eso es de suma importancia lograr que una gran parte de la sociedad tenga la posibilidad de acceder al contenido presente en la web. Aquí es donde la accesibilidad web se convierte en un recurso fundamental para combatir con la famosa brecha digital y permitir que el contenido web sea accesible a la mayor cantidad de personas posibles. Para brindar accesibilidad en un sitio web, es muy importante integrarla al proceso de desarrollo. Para facilitar esta integración y a la detección de problemas de accesibilidad, en esta tesina se desarrolló una herramienta automática para la detección y reporte de este tipo de problemas.

Palabras Clave

Accesibilidad, Usabilidad, Bad Smell, Refactoring, Aplicaciones web, Lector de pantalla, Análisis de accesibilidad, Guías de accesibilidad, ARIA, JavaScript, DOM, API, Extensión Web, Finder, Handler JS, Modal, Formulario Web.

Conclusiones

El trabajo desarrollado permite a los desarrolladores y dueños de aplicaciones web, obtener reportes de problemas de accesibilidad web complejos, que no pueden ser detectados mediante el análisis estático de código. La herramienta resultante es de gran utilidad para mejorar la accesibilidad web, debido a su facilidad para ser extendida con nuevos algoritmos de búsqueda, y a su adaptabilidad para diferentes escenarios de búsqueda.

Trabajos Realizados

- Análisis de herramientas online actuales de accesibilidad web.
- Búsqueda y análisis de problemas de accesibilidad utilizando un lector de pantalla.
- Desarrollo de los algoritmos “finder” para la detección de diferentes tipos de problemas de accesibilidad web.
- Desarrollo de una herramienta para la detección, almacenamiento y reporte de problemas de accesibilidad web, compuesta por una extensión web, una base de datos, una API REST y una aplicación de reportes.
- Realización de pruebas de la herramienta en diferentes sitios web para analizar el cumplimiento de sus objetivos.

Trabajos Futuros

- Ampliar el alcance de la herramienta para poder aplicar refactorings y así poder solucionar problemas de accesibilidad.
- Mejora de la interfaz de la aplicación de reportes: agregar roles de usuarios y mejora de los reportes.
- Escalar el alcance de la herramienta a un ambiente de producción.
- Continuar con la mejora de los finders desarrollados para ampliar el alcance a más tipos de elementos web.
- Continuar con el desarrollo de nuevos finders para otros tipos de elementos web inaccesibles.

Fecha de la presentación: **Agosto 2021**

Agradecimientos

En primer lugar, quiero agradecerles a mis directores de tesina, la Dra. Alejandra Garrido y al Dr. Julián Grigera. Sin su ayuda y dedicación este trabajo no hubiese sido posible. Además de la constante motivación para lograr avanzar y vencer los obstáculos que se fueron presentando.

A mis abuelos Ana y Tito, a mis tíos Ana y José, y a mi hermano Mauricio por acompañarme desde un primer momento y a lo largo de toda mi carrera. Dándome su soporte en todo momento y motivándome para lograr cerrar esta etapa tan importante en mi carrera profesional.

A mis tíos, primos y familia por el apoyo tan valioso para lograr llegar a la meta.

Amigos, compañeros y profesores de la facultad, y colegas del trabajo que siempre se vieron interesados en mis avances y alegrías.

A todos ellos: ¡Muchas Gracias!

Índice

CAPÍTULO 1 6
INTRODUCCIÓN 6
1.1 Motivación 7
1.2 Objetivos 8
1.3 Contribución 9
1.4 Organización de la tesina 10
CAPÍTULO 2 12
TRABAJOS RELACIONADOS Y MARCO TEÓRICO 12
2.1 Accesibilidad 12
2.2 Evaluación de la accesibilidad de un sitio web 18
2.3 Trabajos relacionados 28
CAPÍTULO 3 35
ARQUITECTURA DEL PROYECTO Y TECNOLOGÍAS UTILIZADAS 35
3.1 Introducción 35
3.2 ¿Qué ofrece esta herramienta? 36
3.3 Diagrama de arquitectura 38
3.4 Conceptos utilizados 44
3.5 Diferencias con trabajos relacionados 60
CAPÍTULO 4 62
ANÁLISIS DE ACCESIBILIDAD EN ELEMENTOS WEB INTERACTIVOS CON HANDLERS JS ASIGNADOS 62
4.1 Introducción 62
4.2 Si necesitas un botón, usá el elemento <button> 63
4.3 Ejemplos de elementos inaccesibles con handlers JS asociados 65
4.4 Herramienta para la detección y reporte de elementos interactivos inaccesibles 69
4.5 Resultado final de la herramienta para la detección de elementos inaccesibles con handlers JS 75
CAPÍTULO 5 89
ANÁLISIS DE ACCESIBILIDAD EN MENSAJES DINÁMICOS DE FORMULARIO WEB 89
5.1 Introducción 89
5.2 Motivación del buen uso de notificaciones en formularios 89
5.3 Soluciones de desarrollo para que los mensajes dinámicos sean accesibles 90

5.4 Formularios Web con mensajes de error inaccesibles 91
5.5 Herramienta para la detección y reporte de mensajes dinámicos inaccesibles en formularios web 99
5.6 Resultado final de la herramienta para la detección de mensajes de formularios web inaccesibles 101
CAPÍTULO 6 109
ANÁLISIS DE ACCESIBILIDAD EN ELEMENTOS WEB FLOTANTES 109
6.1 Introducción 109
6.2 Tipos y ejemplos de Non-modal dialogs 110
6.3 Problemas de accesibilidad en elementos flotantes 113
6.4 Consejos de accesibilidad para los non-modal dialog 117
6.5 Herramienta para la detección y reporte de elementos flotantes inaccesibles 120
6.6 Resultado final de la herramienta para la detección de elementos flotantes inaccesibles 125
CAPÍTULO 7 141
CONCLUSIÓN Y TRABAJOS FUTUROS 141
7.1 Conclusiones 141
7.2 Trabajos Futuros 143
CAPÍTULO 8 146
BIBLIOGRAFÍA Y ANEXO 146
8.1 REFERENCIAS BIBLIOGRÁFICAS 146
8.2 ANEXO 148

CAPÍTULO 1

INTRODUCCIÓN

La accesibilidad es una característica de un elemento, lugar, entidad, producto, servicio, página, u aplicación. Es una característica muy especial, que justamente apunta a facilitar el acceso a la mayor cantidad de personas posibles, independientemente de su edad, de sus capacidades, de sus limitaciones y de sus medios técnicos. Teniendo en cuenta la diversidad del contexto en el cual esa persona se sitúa al momento de utilizar ese elemento, o esa entidad, o ese servicio, o ese sitio.

La accesibilidad de alguna manera implica generar independencia para permitir autonomía.

Si bien existen desde hace varios años, entidades internacionales como la W3C [W3C] que se dedican a implementar tecnologías y estándares uniformes para promover la accesibilidad, aún se siguen creando productos y servicios donde no se incluye a la accesibilidad como parte del proceso de desarrollo. Hoy en día la sociedad aún no está muy consciente de lo que significa crear accesibilidad, y debemos encontrarle sentido porque es algo que afecta a mucha cantidad de personas. De hecho, según la OMS (Organización Mundial de la Salud), más de mil millones de personas actualmente tienen algún grado de discapacidad a nivel mundial.

Para lograr aplicar el proceso de generar accesibilidad en un sitio web existen fundamentos y reglas muy claras para lograr este cometido. No es necesario que los desarrolladores sean expertos en la materia, pero deben tratar de interiorizarse y capacitarse en el tema, y así lograr que la mayor cantidad de recursos digitales puedan ser accesibles. Sin embargo, estas capacitaciones llevan tiempo y por lo tanto el costo de lograr accesibilidad aumenta. Esto lleva a que empresas no inviertan en el proceso de accesibilidad de sus productos, y esto produce que el costo de la **no-accesibilidad** sea alto. Entre estos costos existen los contratos extras, se debe contratar servicios externos para la optimización en la mejora de accesibilidad en el sitio web. La tasa de abandono y perdida de usuarios crece con el tiempo, porque se tienen usuarios que entran al sitio web pero no encuentran y no pueden acceder al contenido dentro del mismo. Además, de las sanciones que se pueden llegar a recibir por no proveer un producto accesible.

Existen casos de importancia internacional, donde compañías han sido sancionadas por no proveer accesibilidad en sus sitios web. Por ejemplo, el Departamento de Justicia de Estados Unidos (DOJ), citando la Ley de Estadounidenses con Discapacidades demandó y negoció millones de dólares en acuerdos con grandes marcas, como Target, Disney y Netflix [Digital-Discrimination]:

- En agosto de 2008, en Estados Unidos, después de un juicio de 2 años, la compañía Target se comprometió a pagar a la Federación Nacional de Ciegos una suma de 6 millones de dólares por la falta de accesibilidad de su sitio web.
- Más recientemente, también en Estados Unidos, en febrero de 2011, se presentó una demanda contra la compañía Disney por la falta de accesibilidad en su sitio web.

1.1 Motivación

La accesibilidad web está enfocada en el contenido web, es decir, en el contenido textual, auditivo o visual que un sitio web quiere entregar y facilitar a las personas.

Debido a la gran cantidad de personas que poseen alguna discapacidad, es muy importante crear procesos para proveer accesibilidad en beneficio a estas personas, para que ellos mismos puedan acceder a la mayor cantidad de información y lo más importante, acercar el contenido a la mayor cantidad de personas.

No obstante, no se debe restringir el contexto de accesibilidad a solamente casos extremos, como personas no videntes. Por ejemplo, personas con alguna leve dificultad de visión que utilizan lentes para poder consumir contenido web, pueden estar navegando sitios web accesibles sin la necesidad de utilizar sus lentes. Por lo tanto, no debemos solamente incluir casos extremos de discapacidad cuando se piensa en accesibilidad, sino ser más abarcativos a **todo tipo y grado de discapacidad**.

Generalmente las empresas tratan de agilizar los procesos de desarrollo lo más rápido posible. Por esto, muchas veces se ignoran las cuestiones de accesibilidad y las necesidades de personas discapacitadas que están olvidados y que necesitan también ser parte de esto, de estar al tanto de la tecnología. Por ende, aún se debe seguir trabajando en mejorar la inclusión de accesibilidad en el proceso de desarrollo.

Sin embargo, también existen diferentes productos de software que están en constante crecimiento y su función principal es la de ayudar a personas con algún grado de discapacidad.

Existen las tecnologías de asistencia, donde el principal ejemplo son los screen readers o lectores de pantalla. También están los magnificadores que sirven para aumentar o disminuir en cómo perciben los usuarios el contenido. También existen los asistentes de voz y así se pueden seguir nombrando varias tecnologías más. Además, en los últimos años ha habido un incremento en el control que ofrecen los navegadores a los usuarios para poder controlar la configuración y así lograr adaptar el contenido a sus necesidades.

Sin embargo, estas herramientas pueden no ser suficientemente eficaces para proveer acceso universal en sitios construidos y desarrollados donde no se tuvo en cuenta la accesibilidad desde un principio. Adaptar y modificar estos sitios para que sean accesibles generalmente es un proceso muy costoso, por lo tanto, la solución más económica y eficaz, siempre va a ser la de tener en cuenta desde el principio a la accesibilidad como un requisito más durante el desarrollo.

Para atacar la problemática de mejorar la usabilidad y la accesibilidad en una aplicación existente, se ha propuesto la técnica de **refactoring** [Garrido11], y se desarrolló un framework para implementar estos refactorings en JavaScript llamados Client Side Web Refactorings (CSWR); este nombre se debe a que los mismos se instalan del lado del cliente [Garrido13Pers]. Asociado a los refactorings de usabilidad se definió el concepto de **usability smell** como una indicación de un problema de usabilidad que puede solucionarse a través de refactorings [Garrido11]. De manera similar se definió el concepto de **accessibility smell** [Garrido13Imp]. El principal problema que aún tiene esta tecnología, es que queda reservada para unos pocos que la conocen y puedan extenderla programando nuevos refactorings en JavaScript.

1.2 Objetivos

Muchas veces se cree que desarrollar sitios webs accesibles significa que van a ser sitios más simples, poco atractivos, y con poco contenido debido a restricciones en el diseño, etc. Esto es falso, se pueden seguir construyendo sitios con cualquier tipo de característica que se desee, utilizando las últimas tecnologías de desarrollo web. Simplemente hay que seguir ciertas guías durante el desarrollo, como las famosas Web Content Accessibility Guidelines [WCAG], las cuales son el mejor instrumento que se tiene para guiar a los desarrolladores en la creación de sitios web accesibles.

Lamentablemente, estas guías no garantizan que un sitio sea completamente accesible y cubrir todos los tipos de problemas de accesibilidad que un sitio pueda presentar, pero son un buen punto de partida para lograr el cometido de proveer webs accesibles.

Estas reglas no solo son utilizadas por los desarrolladores, sino también existen herramientas de detección automáticas de problemas de accesibilidad, las cuales ayudan a los programadores a identificar violaciones a las pautas detalladas en las WCAG. Por ejemplo, la herramienta llamada TAW, tiene como objetivo comprobar el nivel de accesibilidad alcanzado en el diseño y desarrollo de páginas web con el fin de permitir el acceso a todas las personas independientemente de sus características diferenciadoras [TAW].

Sin embargo, estas herramientas tienen sus limitaciones, ya que realizan un **análisis estático** a nivel de código. Por lo tanto, la cantidad de problemas que pueden encontrar es limitada. Para esta problemática, se han ido desarrollando nuevas herramientas que realizan **búsquedas dinámicas** analizando los patrones de interacciones generados por los usuarios.

Entre estas soluciones incluimos a la herramienta denominada Kobold, la cual detecta automáticamente dificultades de usabilidad en aplicaciones web reportando sus posibles soluciones basadas en acciones que los usuarios realizan sobre la interfaz [Grigera17]. En su desarrollo se adaptaron los términos del **refactoring** de código definiendo problemas de usabilidad como **usability smells** a partir de los cuales puede aplicar soluciones que denomina **usability refactorings**.

También se ha desarrollado una versión de Kobold “accesible”, la cual parte de la arquitectura base de Kobold para usabilidad, logrando implementar una extensión para procesar interacciones que incluyan eventos de accesibilidad[Durgam20]. En este trabajo también se definió un catálogo de **accessibility smells**, los cuales están asociados con dificultades visuales que puedan detectarse automáticamente al utilizar el teclado con un lector de pantalla. Como también, un catálogo de **accessibility refactorings** que aplican transformaciones a la interfaz como solución a la presencia de un accessibility smells.

Esta tesina busca continuar con la iniciativa de desarrollar nuevas herramientas automáticas, que logren permitir a los dueños y desarrolladores de sitios webs, poder cumplir con los niveles más altos de accesibilidad. Asegurando la detección y reporte de un amplio catálogo de problemas de accesibilidad.

Para lograr esto, en esta tesina se tiene como objetivo general proponer el desarrollo una herramienta automática para la detección de bad smells de accesibilidad.

Resumiendo, los objetivos son:

- Encontrar y reconocer los patrones de comportamiento de elementos webs específicos, que sean indicadores de que hay problemas en la interacción con la aplicación web.
- Poder reportar problemas de accesibilidad que ocurren a partir de los eventos de interacción generados por el usuario.
- Desarrollar una herramienta que detecte automáticamente los smells de accesibilidad que se logren identificar, como extensión a herramientas existentes de detección de smells de usabilidad.
- Permitir al dueño o a desarrolladores, acceder a un reporte detallado de bad smells de accesibilidad presentes en su sitio web.

1.3 Contribución

Este trabajo tiene la finalidad de poder detectar problemas de accesibilidad web complejos, mediante la utilización de diversas técnicas que provee el lenguaje JavaScript, para lograr detectar diferentes tipos de elementos web que fueron desarrollados de forma que sean inaccesibles. Estos elementos generan problemas de accesibilidad que debido a su complejidad no está incluida su detección en herramientas de detección automáticas actuales.

Partiendo de varios trabajos de investigación previos y tomando como punto de partida las herramientas desarrolladas en dichos trabajos, se construyó una herramienta de detección automática para problemas de accesibilidad, que consiste en:

- ✓ Realizar búsquedas de *bad smells* de accesibilidad en tiempo real, dentro del navegador web mientras el usuario interactúa con diferentes sitios web. Esto se logra gracias a una extensión web, desde la cual se puede acceder a la estructura del DOM de la página web actual. La misma permite definir finders o “buscadores” para diferentes tipos de problemas de accesibilidad.
- ✓ Una serie de algoritmos de búsqueda de bad smells de accesibilidad, los cuales denominamos para esta tesina como **finders**. Cada uno destinado a la búsqueda de un smell particular, y debido a cómo se definió la arquitectura de la herramienta, se pueden agregar más finders cuando se requiera.
- ✓ Una *API REST* para recibir toda la información recolectada por la extensión web. La misma está conectada a una base de datos para almacenar toda esta información de una forma estructurada, para luego poder consumirla cuando sea necesario.
- ✓ Se desarrolló una aplicación web para poder visualizar la información obtenida por la web extensión, pero en una forma estructurada para lograr una mejor organización de los datos y así poder presentarla en un formato de tipo reporte.

Además, en este trabajo se suman las siguientes contribuciones:

- ✓ Se enumeraron diferentes herramientas web automáticas de análisis estático para accesibilidad web. Se estudiaron y realizaron diferentes pruebas en varios sitios web, para mostrar los problemas de accesibilidad detectados y sus carencias en cuanto a estas búsquedas.

- ✓ Se estudiaron diferentes métodos y herramientas que no sólo se centran en el análisis estático de código, sino que llevan el nivel de búsqueda a una complejidad más alta. No solo ofrecen la detección de problemas sino también poder aplicar una solución directamente.
- ✓ Además del desarrollo de una nueva herramienta, se realizaron varias pruebas de la misma en diferentes sitios web. Esto permite demostrar la eficacia de la misma y enumerar los casos específicos de elementos web o *smells* que no han podido ser detectados.

1.4 Organización de la tesina

- ❖ **Capítulo 1, Introducción:** en este capítulo se presenta la motivación, objetivos, contribución y la organización de la tesina.
- ❖ **Capítulo 2, Trabajos Relacionados y Marco Teórico:** en este capítulo se introducirá al lector dentro del contexto del trabajo, donde se describen los conceptos básicos necesarios para comprender este trabajo y, además, se describen brevemente aquellos trabajos que motivaron la realización de esta Tesina.
- ❖ **Capítulo 3, Arquitectura del Proyecto y Tecnologías Utilizadas:** en este capítulo se describe como está constituida la arquitectura de la herramienta desarrollada para esta tesina. Se describe cada uno de los componentes que la integran y cuál es la función de cada uno. También se describen varios de los conceptos más importantes utilizados a lo largo de esta tesina, y las diferencias de este desarrollo con otras propuestas similares.
- ❖ **Capítulo 4, Análisis de Accesibilidad en Elementos Web interactivos con Handlers JS Asignados:** en este capítulo se describe el método utilizado para la detección de los elementos interactivos web inaccesible que son desarrollados utilizando handlers JavaScript. Se describe la problemática, varios casos de ejemplo, junto a la implementación del *finder* para la búsqueda de estos elementos, como también varios casos donde se ejecutó la herramienta junto a sus respectivos resultados.
- ❖ **Capítulo 5, Análisis de Accesibilidad en Mensajes Dinámicos de Formularios:** en este capítulo se describe el método utilizado para la detección de los mensajes de errores en formularios que son inaccesibles. Se describe la problemática, varios casos de ejemplo, junto a la implementación del *finder* para la búsqueda de estos elementos, como también varios casos donde se ejecutó la herramienta junto a sus respectivos resultados.
- ❖ **Capítulo 6, Análisis de Accesibilidad en Elementos Web Flotantes:** en este capítulo se describe el método utilizado para la detección de elementos web flotantes que presenten problemas de accesibilidad. Se describen los diferentes tipos de elementos flotantes que pueden estar presentes en un sitio web y los problemas que estos pueden presentar. Además, se describirá el *finder* desarrollado en la herramienta para este tipo de elementos, junto con varios ejemplos en diferentes sitios web, como también la ejecución de la herramienta y los reportes que esta genera.

- **Capítulo 7, Conclusión y Trabajos Futuros:** Este capítulo resume a modo de conclusión las contribuciones de esta tesina, así como las diferentes propuestas de trabajos futuros.
- **ANEXO**
  - **Cómo ejecutar la herramienta:** en este anexo se describen los pasos a seguir para poder instalar y utilizar la herramienta desarrollada para esta tesina en forma local.

CAPÍTULO 2

TRABAJOS RELACIONADOS Y MARCO TEÓRICO

2.1 Accesibilidad

La **accesibilidad** es la medida en que los productos, sistemas o servicios, entornos e instalaciones, pueden ser utilizados por personas de una población con la más amplia gama de necesidades, características y capacidades de los usuarios para lograr los objetivos identificados en contextos de uso identificados.

2.1.1 Usabilidad

La **usabilidad** es la medida en que un sistema, un producto o un servicio puede ser utilizado por usuarios específicos para lograr objetivos específicos, con efectividad, eficiencia y satisfacción, en un contexto de uso específico [ISO11].

Es muy importante que nos fijemos en que la palabra específico aparece repetida tres veces: “usuarios específicos para lograr objetivos específicos en un contexto de uso específico”.

Por lo tanto, cuando hablamos de usabilidad tenemos que tener definidas a estas tres dimensiones. Tenemos que definir de qué usuarios estamos hablando por ejemplo por el rango de edad, si estamos hablando de usuarios entre 20 y 30 años o usuarios adultos mayores entre 60 y 70 años, por ejemplo. Para qué objetivo queremos que usen el producto y el contexto de uso dónde o cuándo lo van a usar.

2.1.2 Accesibilidad y Usabilidad

Si bien la usabilidad y la accesibilidad son dos conceptos que están íntimamente relacionados a veces se solapan entre sí, ya que los dos buscan lograr que el usuario obtenga la mejor experiencia cuando usa un producto o un servicio.

Hay una diferencia importante en lo que respecta al usuario. En la usabilidad el usuario es específico, o sea que **debemos definir a qué usuarios nos estamos refiriendo**.

Sin embargo, en la accesibilidad se apunta a cualquier persona de una población con la más amplia gama de necesidades, características y capacidades. Es decir, la accesibilidad busca que algo pueda ser usado por todo el mundo y esto significa que el producto, el servicio o la instalación pueda ser usado incluso por las personas con discapacidad.

Resumidamente, podemos decir que esta es la diferencia más importante, donde la usabilidad se refiere a unos usuarios concretos pero la accesibilidad se refiere a todos los posibles usuarios, incluyendo nuevos usuarios con algún tipo de discapacidad

Entonces como se definió anteriormente, la usabilidad busca la facilidad de uso para usuarios concretos, en un contexto de uso concreto, para un objetivo concreto o específico como por ejemplo una norma ISO. Pero la accesibilidad lo que busca es la posibilidad de uso para que todo el mundo pueda usar este producto, ese servicio, esa instalación.

Un sinónimo que se suele emplear con accesibilidad o un término relacionado, es el concepto de diseño inclusivo, diseño universal y diseño para todos.

La accesibilidad y usabilidad son conceptos que están estrechamente ligados. Por ejemplo, cuando planteemos el diseño y el desarrollo de un producto o un servicio en una instalación de un edificio. El primer paso sería lograr la accesibilidad para que todos los usuarios lo puedan usar y aprovechar. El siguiente paso debería ser lograr que lo puedan usar con la mayor facilidad de uso, por lo tanto, en un primer lugar nos debemos preocupar de la accesibilidad y en un segundo paso de la usabilidad.

2.1.3 Accesibilidad en aplicaciones web

La accesibilidad web hace referencia a la capacidad de acceso a un sitio web por todo tipo de usuarios independientemente de sus limitaciones de modo que los usuarios sean capaces de percibir, entender, navegar, e interactuar con dicho sitio de forma satisfactoria.

Un documento web debería poder ser accesible para todos, excepto para los que tienen alguna discapacidad cognitiva que les impida la comprensión del texto. Por lo tanto, deberíamos poder percibirlo y comprenderlo por sí mismo, y poder utilizar todos sus elementos y podamos navegar por él.

Por ejemplo, no sirve de nada un formulario web que no podemos rellenar o un documento web con gran cantidad de hojas que no contiene un índice que permita movernos por todos sus apartados.

La interacción con las computadoras es realizada a través de una **interfaz**, esta comprende el **punto de contacto entre el humano y la información** dispuesta en los medios tecnológicos [María and Paz 2012]. Es por esto que las interacciones con dichas interfaces comprenden el principal objeto de estudio para la mejora de la accesibilidad.

¿Qué sucede con la usabilidad web?

Para algunos la accesibilidad es un término de la usabilidad, para otros de lo contrario, la usabilidad es parte de la accesibilidad [Caballero2012]. En la práctica **se incluyen mutuamente y en ningún caso se excluyen**, ya que el diseño accesible debe ser usable y de hecho si se aplican correctamente las directrices de accesibilidad se obtendrá un diseño usable. Estos dos términos están íntimamente relacionados ya que ambas mejoran la satisfacción, la efectividad y deficiencia de los elementos para todos, con o sin discapacidad.

2.1.4 Problemas por falta de accesibilidad web

Normalmente, durante la navegación web, los usuarios videntes no suelen leer todas las palabras presentes en las páginas para comprenderlas, sino que realizan saltos continuos, escaneando las distintas partes de las páginas para concluir si es importante o no, si se trata de la información que buscaban o si precisan dirigirse hacia otro sitio [Nielsen06].

Esto no es muy distinto en el caso de los usuarios con disminución visual, si bien ellos utilizan lectores de pantalla que leen el texto presente en las interfaces, son igual de impacientes a la hora de obtener lo que buscan lo más rápido posible. Al utilizar los lectores de pantalla no escuchan cada palabra de una página, escuchan solo las primeras palabras de un enlace o una línea de texto y si no les parece relevante se desplazan al siguiente [Peter Krantz 2005]. A pesar de esto, interactuar mediante el uso de un lector de pantalla trae consigo varios problemas en la navegación:

Falta de contexto: Al utilizar un lector de pantalla, el usuario puede perder la noción del contexto general en donde el puntero del lector se encuentra ubicado. El mismo continuará pronunciando lecturas de texto de las cuales el usuario ya no comprende de donde provienen. (Ejemplo: Wikipedia suele utilizar varios links dentro del contenido principal de los documentos. Si navegamos link a link usando lectores de pantalla, escucharemos palabras aisladas las cuales resultarán confusas y fuera de contexto. Si bien es cierto que los lectores suelen avanzar línea por línea o de a párrafos, aun así, es muy fácil desorientarse y perder conocimiento de la ubicación del lector en la página web).

Lectura secuencial: Los comandos para la navegación pueden obligar al usuario a recorrer el contenido de la aplicación de manera secuencial, por lo tanto, es muy importante introducir mecanismos que faciliten la identificación precisa de las partes que comprender a la página y de dar mayor importancia a aquellas que contengan el contenido por el cual se está ingresando. (Ejemplo: en los resultados de búsquedas, la lectura secuencial puede trazar un camino por publicidades y links innecesarios antes de llegar a los verdaderos resultados buscados).

Estos problemas representan tanto falta de accesibilidad como de usabilidad: El usuario posee acceso a gran parte de la información textual y/o sonora, pero carece de acceso a la información visual, la cual atenta en contra de la orientación y dificulta la navegación y el uso de la aplicación. Cuando la aplicación posee una baja usabilidad gran parte de la información no es accedida por falta de interpretación. Es decir, que la información tal vez pueda ser leída por un lector de pantalla, pero la falta de contexto y la dificultad para manejarlo atenta contra la interpretación de dicha información haciéndola prácticamente inaccesible.

Aunque la accesibilidad y la usabilidad se relacionan en gran medida, muchas veces son tratadas por separado. La accesibilidad tiene como objetivo hacer los sitios web disponibles para la mayor cantidad de personas posibles, mientras que la usabilidad se centra en mejorar la experiencia del usuario en el sitio, a modo de que esta sea más eficiente y satisfactoria. Como mencionamos anteriormente, problemas en la accesibilidad concluyen en problemas de usabilidad y viceversa. Por lo tanto, si bien nuestro objetivo es mejorar la accesibilidad de los usuarios a las páginas web, gran parte de ello lo lograremos mejorando la usabilidad de las mismas.

Por último, los programadores web a menudo no tienen conciencia sobre los estándares de accesibilidad, deciden ignorarlos o aplican conceptos vagos [Ball 2013], ya que consideran innecesaria la inversión de su tiempo y esfuerzo en lo que puede significar un público pequeño con necesidades particulares. Por lo tanto, existen muchas páginas web que no brindan una alternativa para la navegación de quienes deben utilizar lectores de pantalla.

Esto genera que el uso de lectores de pantalla no sea de lo más eficiente y los usuarios de los mismos no puedan utilizar adecuadamente toda la funcionalidad de la aplicación web. De estas funcionalidades, hay varias que están implementadas utilizando técnicas inadecuadas o que generan componentes que no cumplen con los estándares de accesibilidad, logrando que estos sean inaccesibles para los usuarios de screen readers. Entre los casos más destacados encontramos:

- Listas desplegables en los cuales no se pueden acceder a sus componentes.

- Botones de envío de formularios implementados con o <a>(links), los cuales en varias ocasiones son ignorados por el screen reader, como también no funcionan al intentar accionarlos con la tecla ENTER.
- Utilización de funciones JS para darle funcionalidad a un elemento determinado, donde muchas veces como en el caso anterior no pueden ser accesibles por usuarios de screen readers.

2.1.5 Ventajas de las páginas web accesibles.

Algunas de las ventajas más importantes que logramos al tener un sitio web accesible son:

- Tenemos un incremento de audiencia web, que es la maximización de los usuarios potenciales. Ósea, logramos un mayor alcance de la comunicación entre servicios o mercados ayudando a reducir la denominada brecha digital.
- Mayor eficiencia y tiempo de respuesta, ya que las páginas están limpias de código inútil o poco eficiente. Su tamaño es menor, por lo que el tiempo de carga es mucho menor.
- Reducción de costes de desarrollo y mantenimiento, ya que una página web accesible es una página bien hecha. Es menos propensa a contener errores y más sencilla de actualizar.

2.1.6 Garantizar accesibilidad web

La accesibilidad Web se ha entendido siempre como responsabilidad de los desarrolladores Web. No obstante, el software Web tiene también un papel importante en la accesibilidad Web. Es importante que el software ayude a los desarrolladores a generar y evaluar sitios Web accesibles para que las personas con discapacidad puedan utilizarlos.

Una de las funciones de la Iniciativa de Accesibilidad Web (WAI) es desarrollar pautas y técnicas que proporcionen soluciones accesibles para el software Web y para los desarrolladores Web. Las pautas de WAI son consideradas como estándares internacionales de accesibilidad Web.

2.1.7 W3C y WCAG

A nivel internacional el W3C [W3C] (un grupo independiente que define los protocolos y estándares para la web) lanzó una de sus iniciativas llamada WAI (Web Accessibility Initiative) [W3C97], la cual estableció diferentes pautas o guías que explican cómo se tienen que crear las páginas web para que sean accesibles. Estas guías son llamadas **WCAG** (Web Content Accessibility Guidelines) de las cuales existen las versiones 1.0, 2.0, y actualmente se encuentran en la versión 2.1 [WCAG]. Estas pautas son las que generalmente suelen ser tomadas en cuenta por la mayoría de los organismos que buscan mejorar los niveles de accesibilidad.

Las Pautas de Accesibilidad al Contenido en la Web 1.0 (WCAG 1.0) fueron aprobadas en mayo de 1999 y es una versión estable y de referencia. Sin embargo, las WCAG 2.0 han sido desarrolladas para aplicarse a diferentes tecnologías y, a su vez, para que su utilización y compresión sea sencilla, y para que su comprobación sea más precisa.

WCAG 1.0 utiliza **15 pautas**. Cada pauta tiene varios checkpoints, y cada checkpoint tiene un nivel de accesibilidad (A, AA o AAA).

WCAG 2.0 se divide en **4 grandes principios**. Cada principio tiene varias pautas, y cada pauta 3 niveles de éxito. Cada nivel de éxito tiene varios puntos a cumplir.

En nivel 1 de éxito de la WCAG 2.0 equivale al nivel A de la WCAG 2.0, el nivel 2 a la AA y el 3 a la AAA.

Los cuatro principios básicos de la WCAG 2.0 son:

- El contenido debe ser perceptible.
- Los elementos de la interfaz en el contenido deben ser manejables.
- El contenido y los controles deben ser comprensibles.
- El contenido debe ser suficientemente robusto para funcionar con las tecnologías actuales y las del futuro.

Una ventaja importante es que en las WCAG 2.0, cada guía va acompañada de un párrafo explicando a qué tipo de usuarios ayuda y con varios ejemplos de uso.

WCAG 2.0 se publicó el 11 de diciembre del 2008. WCAG 2.1 se publicó el 5 de junio del 2018, incluyendo 17 criterios nuevos.

El objetivo de esta nueva versión de las WCAG (2.1) es mejorar las pautas de accesibilidad para tres grupos específicos de usuarios:

- Las personas con discapacidad cognitiva o del aprendizaje.
- Las personas con baja visión.
- Las personas con discapacidad que acceden desde dispositivos móviles.

Aunque se mejora en estas áreas, se subraya que no cubren todas las necesidades de estos usuarios.

De cualquier modo, siguiéndolas, se hará la Web más accesible también para todos los usuarios, cualquiera que sea la aplicación que el usuario esté utilizando, por ejemplo: computadora de escritorio, navegador de voz, teléfono móvil, central multimedia de automóvil, etc.; o las limitaciones bajo las que opere, por ejemplo: entornos ruidosos, habitaciones infra o supra iluminadas, entorno de manos libres, etc. Seguir estas pautas ayudará también a que cualquier persona encuentre información en la Web más rápidamente. Estas pautas no limitan a los desarrolladores en la utilización de elementos y componentes multimedia como imágenes, vídeo, etc., sino al contrario, explican cómo hacer los contenidos multimedia más accesibles a una amplia audiencia.

2.1.8 Lectores de pantalla

Los usuarios discapacitados con problemas de visión utilizan lectores de pantalla para acceder al contenido. Por ejemplo: el lector de pantallas open source llamado NVDA. Su propósito es permitir la utilización del sistema operativo y las distintas aplicaciones mediante el empleo de un sintetizador de voz que "lee y explica" lo que se visualiza en la pantalla, lo que supone una ayuda para las personas con graves problemas de visión o completamente ciegas.

NVDA es un proyecto de software libre, por lo que también está disponible el código fuente del programa de forma gratuita [NVDA]. Puede ser extendido a través de complementos.

El desarrollo de complementos para NVDA, es una forma de extender el comportamiento de la aplicación dotándola de nuevas y/o mejores características.

Por ser software open source, los desarrolladores tienen a su disposición la jerarquía completa de clases y APIs que facilitan adicionar paquetes con funciones y controles ajustados a sus necesidades específicas.

Durante el periodo de evaluación, desarrollo y testing de esta tesina se utilizó como screen reader principal a NVDA, así que será nombrado repetidas veces a lo largo de este documento.

2.1.9 Bad accessibility smells

Cuando se plantea el desarrollo de aplicaciones Web accesibles, el objetivo del diseño es cumplir con la Guía de Accesibilidad para el Contenido Web (WCAG Web Content Accessibility Guidelines) de la W3C. Sin embargo, y a pesar, de cumplir con las recomendaciones de la W3C, la aplicación puede no ser usable o accesible, debido a que personas con discapacidades podrían aun tener dificultades para navegar e interactuar con el contenido de manera cómoda, fácil y eficaz. Esto se da sobre todo en las aplicaciones con contenido dinámico, llamadas Rich Internet Applications (RIA), que cambian la página web y el estado de los widgets en la página a medida que el usuario interactúa con ella. Estas interacciones no son evaluadas por ninguna herramienta automática, dado que las herramientas solo chequean y analizan el código HTML estático como veremos en la sección 2.2.

Una propuesta para solucionar problemas de usabilidad o accesibilidad que aparecen en el contenido estático o a medida que se interactúa dinámicamente con una página web es el concepto de **Refactoring Web** [Garrido11]. El Refactoring originalmente asociado con la reestructuración de código fuente, es ahora aplicado, con la misma idea de reestructuración, al cambio del “look & feel” de una aplicación web.

Con estos refactorings los desarrolladores pueden cambiar aspectos de la presentación de una aplicación web actualmente en funcionamiento, el cual pruebe ser inadecuado o difícil de usar. Estos refactorings se los reconoce como Refactoring de Interface Web (WIRs Web Interface Refactorings) y pueden ser aplicados del lado del cliente, denominándose Client Side Web Refactorings (CSWR) [Garrido13Pers].

La literatura de los refactorings llama “**bad smells**” a los signos de problemas que degradan la calidad interna del código. En el caso de la accesibilidad, se denominan a estos problemas como “**bad accessibility smells**”. Estos son detectados generalmente por del usuario a través de su interacción con la aplicación web [Garrido13Imp] [Durgam20]. Otras fuentes para estos “smells”, emergen por el pobre o no cumplimiento con las guías de accesibilidad WCAG [WCAG] y la tecnología WAI-ARIA [WAIARIA].

Los bad accessibility smells señalan deficiencias en la interfaz que pueden ser detectadas y corregidas por los desarrolladores.

2.2 Evaluación de la accesibilidad de un sitio web

Cuando se está construyendo una aplicación web, el objetivo del diseño es lograr la conformidad con las pautas WCAG, organizados según su nivel de cumplimiento asociado: A (nivel bajo), AA (medio) y AAA (alto). Durante el desarrollo o rediseño de un sitio Web, la evaluación de la accesibilidad de forma temprana y a lo largo del desarrollo permite encontrar al principio problemas de accesibilidad, cuando es más fácil resolverlos. Técnicas sencillas, como es cambiar la configuración en un buscador, pueden determinar si una página Web cumple algunas de las pautas de accesibilidad. Una evaluación exhaustiva, para determinar el cumplimiento de las pautas, es mucho más compleja.

2.2.1 Evaluación estática vs. Dinámica

Hay herramientas de evaluación online que ayudan a realizar evaluaciones de accesibilidad, de las cuales vamos a detallar más adelante en la sección 2.2.2, tenemos a Level Access, TAW, AChecker, OAW entre otras. No obstante, ninguna herramienta en sí misma puede determinar si un sitio cumple o no las pautas de accesibilidad. Para determinar si un sitio Web es accesible, es necesaria la evaluación humana [W3C05]. Para lograr entender mejor este concepto continuación describiremos varios problemas que no pueden encontrarse mediante búsquedas estáticas en el código de la página web.

Entre los problemas de accesibilidad más recurrentes entre varias aplicaciones webs testeadas, se encuentran los siguientes:

- Mala implementación de elementos interactivos (botones, listas desplegables, etc.) dentro de un formulario.

  En el caso del portal de autogestión de APR, existe un botón que despliega un formulario, Figura 2.2.1, en el cual hay un *select list* sin ningún tipo de atributo explicando que es un listado desplegable, por lo tanto, el usuario puede necesitar realizar varios recorridos dentro del formulario para entender el funcionamiento del elemento, ya que además presionando la tecla ENTER no despliega el menú, sino que el usuario debe ubicarse sobre el elemento y moverse entre los elementos del mismo con las teclas de arriba y abajo. Además, en este ejemplo el formulario incluido en un modal puede no estar presente en el código de la aplicación al momento de hacerse el chequeo estático, ya que este tipo de elementos se generan dinámicamente mediante la interacción del usuario. Ejemplo:

  https://autogestion.apronline.gov.ar/

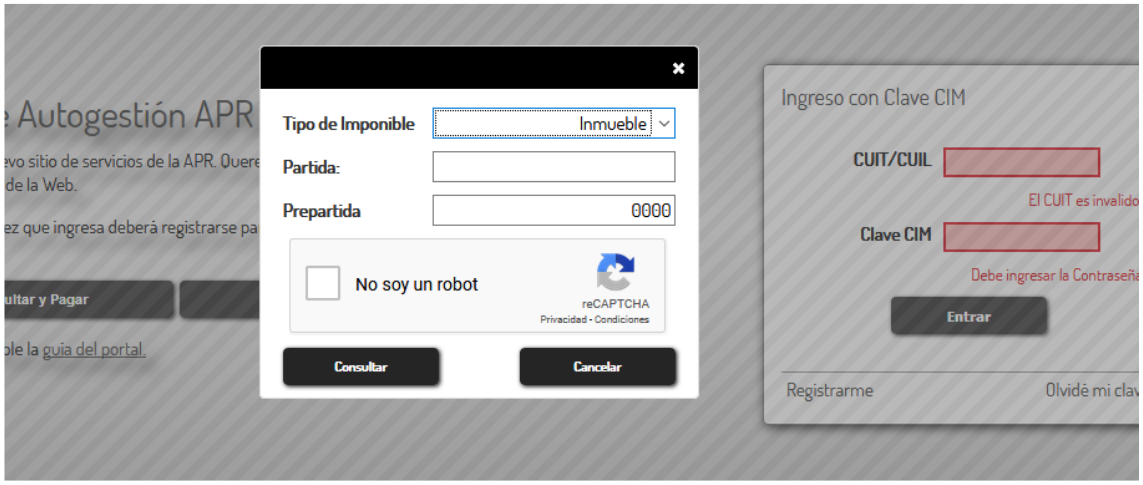


Figura 2.2.1: Modal con formulario del Portal de APR.

- Otro caso relacionado es el de los botones hechos con elementos HTML <a> (para definir links) o (para agrupar contenido), en el cual el usuario quiere buscar estos botones sin poder encontrarlos, lo cual lleva a que realice varios recorridos dentro de la misma sección o página completa para lograr ubicarlos.

Estos defectos pueden no ser detectables por métodos de **búsqueda estáticos**, observando el código de la aplicación web, como en los ejemplos que veremos a continuación en la sección 2.2.2, sino que son generados por la interacción del usuario y se necesita una herramienta que realice **búsquedas dinámicas** observando estas interacciones y reporte los posibles bad smells presentes en la app web.

La idea en principio para la implementación de la herramienta desarrollada en esta tesina, es poder realizar estas búsquedas dinámicamente en busca de posibles problemas de accesibilidad, observando la interacción del usuario en la aplicación web, reportar los elementos de la página involucrados, luego realizar la búsqueda estática dentro del código parseado y buscar los posibles bad accessibility smells dentro del código. Esto va a lograr que se focalice la búsqueda en una cierta sección de la página, o en archivos JavaScript de la misma ya que en estos se encuentran funciones JS asociadas a diferentes elementos para darle funcionalidad a los mismos, esta es una técnica muy común dentro de muchas aplicaciones webs actuales, generando problemas de accesibilidad a los usuarios que utilizan lectores de pantallas para navegar e interactuar con las aplicaciones.

2.2.2 Herramientas de evaluación de accesibilidad web

Level Access

Es una empresa que provee soluciones informáticas, software, consultoría y capacitación para mejorar la accesibilidad en aplicaciones web para empresas corporativas, gubernamentales y de educación. Estas soluciones aseguran conformidad con las leyes, standars y guidelines de accesibilidad actuales [LEVELACC].

Level Access ofrece la herramienta AMP (Accessibility Management Platform), es un centro de comando para la organización, que resuelve la complejidad y los desafíos de la administración de toda la empresa para que pueda construir, probar y mantener software accesible de manera rápida y eficiente [LEVELACC]. Ofrece reportes muy completos sobre los problemas de accesibilidad encontrados en dicha aplicación. Estos reportes contienen descripción de los problemas presentes y cuáles podrían ser sus posibles soluciones, como se puede ver en la Figura 2.2.2.

AMP no aplica soluciones a los problemas de accesibilidad encontrados, simplemente se ejecutan herramientas automáticas de detección de problemas “estáticos”, mientras que los problemas “dinámicos” son encontrados y reportados por personas a cargo de detectar este tipo de problemas.

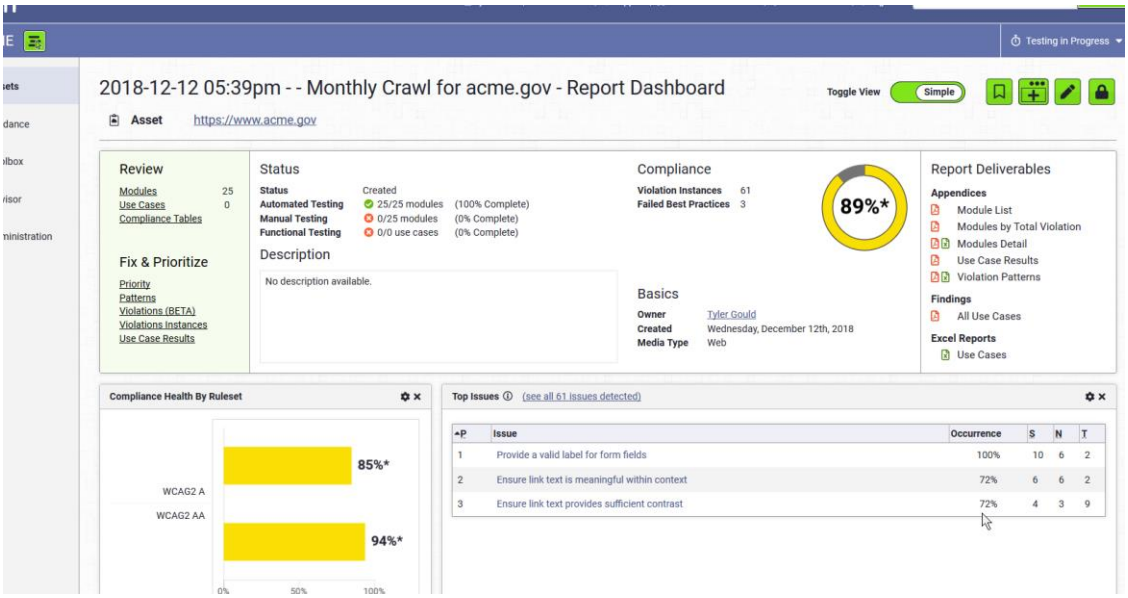


Figura 2.2.2: Dashboard con el reporte generado por AMP para el sitio acme.gov

Los problemas que reporta AMP generalmente son problemas que también detectan otras herramientas, (TAW, AChecker, OAW) ya que son problemas “estáticos” que se buscan analizando el código HTML de la página.

La principal diferencia entre Level Access y los demás validadores mencionados, es que es un servicio pago, ofrece diferentes paquetes y precios dependiendo el tamaño del sistema o empresa. Este servicio incluye tanto la validación por parte de las herramientas automáticas desarrolladas por la empresa, sino también el análisis de personal experto en accesibilidad web. Esto logra que los reportes finales no solo se listen los problemas estáticos encontrados por las herramientas, además van a ser detectados problemas que solo pueden ser hallados por personas debido a su complejidad. Tal cual observamos en la figura 2.2.3, vemos que tiene una lista de estados de casa uno de los tipos de test que se realizan sobre la aplicación web. Los test automatizados son aquellos ejecutados por las herramientas de búsqueda automáticos, mientras que los test manuales son ejecutados por personas con experiencia en este tipo de testeo. Mientras que los test funcionales hacen referencia a la ejecución de casos de uso para analizar distintas funcionalidades del sitio web, también realizadas por personas profesionales.

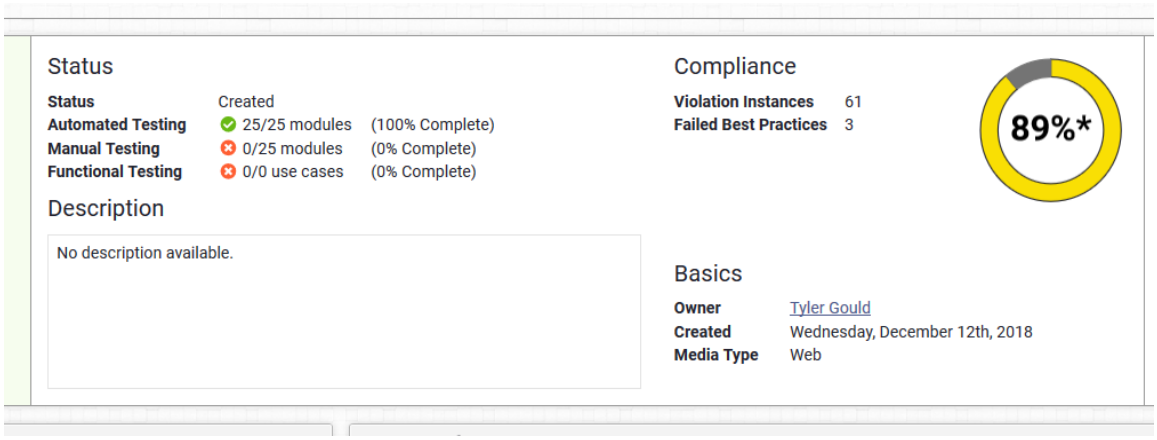


Figura 2.2.3: Sección donde se muestran los estados de los diferentes tipos de test de accesibilidad.

A continuación, daremos algunos ejemplos de problemas de accesibilidad que pueden ser encontrados en los reportes de AMP.

Alguno de los problemas de accesibilidad detectados **estáticamente**:

- No uso de elementos títulos(headings): Los tag <h1>, <h2>, … ayudan a estructurar el texto que contiene el sitio web. Además, son de gran utilidad para los screen readers ya que estos pueden ir navegando el contenido a través de los títulos.
- Sin texto ALT en imágenes.
- Contraste de color insuficiente.
- Sin presencia de roles y etiquetas ARIA.
- Asegurarse que el foco del teclado esté indicado visualmente.
- Asegurarse que los botones no tengan la misma descripción, ej: dos botones con texto “Enviar”. Genera confusión para usuarios de screen readers.
- Proveer de descripciones a los inputs de formularios, utilización de *fieldsets* para lograr esto.

Problemas localizados por **detección humana**:

- Asegurarse que el orden de lectura del contenido y orden del foco sean lógicos: Los usuarios de screen reader suelen perderse si el orden del contenido no es lógico.
- El contenido de la página no debería evitar al usuario de poder cambiar la orientación del display. Y durante este cambio de orientación de pantalla, todo el contenido debe continuar siendo visible y accesible.
- Cuando el usuario reajusta el tamaño de la fuente, no debe haber perdida de contenido y funcionalidad. Usuarios con baja visión no podrían leer el contenido si no logran agrandar el tamaño de fuente.
- Todas las paginas deben tener un título informativo que resuma el contenido de la página. Los usuarios de screen reader podrían no identificar rápidamente cual es el sentido de la página si el título no es informativo.

- Textos de links y botones, no informativos y redundantes. Varios screen readers proveen la funcionalidad de armar un listado de todos los links de una página así el usuario puede navegar solamente entre ellos, por ende, estos links deben tener un texto que represente el contexto y uso del elemento.

Taw

TAW es una herramienta web para el análisis e información del grado de accesibilidad que presentan otras webs. El objetivo de esta web, es difundir la accesibilidad como requisito en el diseño y realización de páginas web con el fin de permitir el acceso a todas las personas [TAW]. Fue creada teniendo como referencia técnica las pautas de accesibilidad al contenido web (WCAG 2.0) del W3C [W3C]. El objetivo de TAW es comprobar el nivel de accesibilidad alcanzado en el diseño y desarrollo de páginas web con el fin de permitir el acceso a todas las personas independientemente de sus características diferenciadoras [TAW].

Al igual que AMP de LevelAccess, las comprobaciones que realiza TAW en el análisis se dividen en dos categorías:

- **Automáticos**. Problemas de accesibilidad que la herramienta detecta por si sola y que deben ser solucionados.
- **Manuales**. La herramienta señala la existencia de un posible problema que el evaluador/a debe confirmar o descartar.

TAW clasifica a los *bad smells* de accesibilidad en las siguientes categorías:

Perceptible

La información y los componentes de la interfaz de usuario deben ser presentados a los usuarios de modo que puedan percibirlos.

Operable

Los componentes de la interfaz de usuario y la navegación deben ser operables.

Comprensible

La información y el manejo de la interfaz de usuario debe ser comprensible.

Robusto

El contenido debe ser suficientemente robusto como para ser interpretado de forma fiable por una amplia variedad de agentes de usuario, incluyendo las ayudas técnicas.

Se probó esta herramienta en varias aplicaciones web. Los cuales solo mencionaremos uno, Figura 2.2.4, y en la Tabla 2.2.2 daremos el resultado total de errores encontrados.

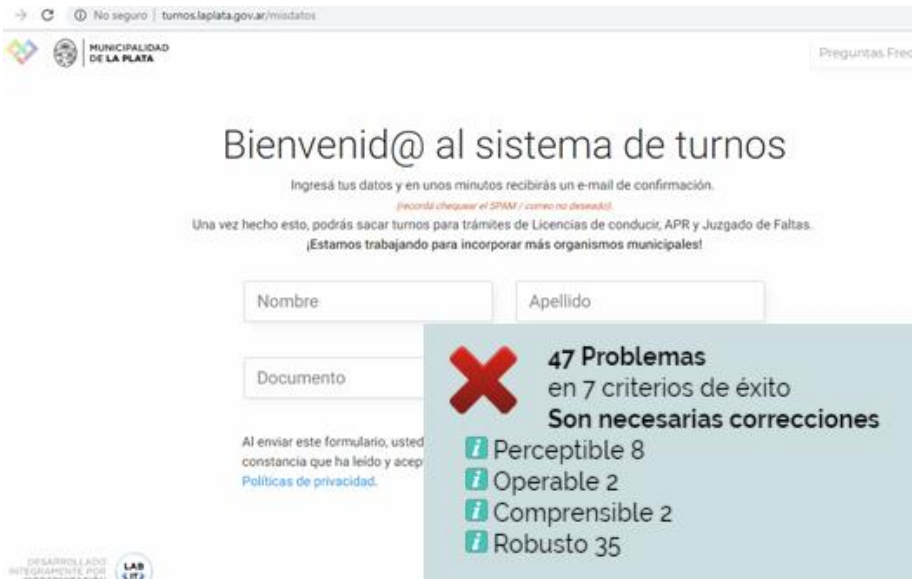


Figura 2.2.4: Captura del sitio web para solicitar turnos en la ciudad de La Plata. Junto al resultado dado por TAW.

Problema	Cantidad
Controles de formulario sin etiquetar.	1
Imágenes sin atributo ALT	3
Enlaces consecutivos de texto e imagen al mismo recurso	1
Inexistencia de elemento h1	1
Encabezados vacíos, sin texto en su interior	1
Enlaces sin contenido	2
Declaración de idioma del documento	1
Página 'bien formada'	33
Marcos sin título	1

Tabla 2.2.1: Resultado de la ejecución de TAW sobre el sitio web de Turnos La Plata.

En la Tabla 2.2.1, vemos el resultado de la ejecución de TAW sobre el sitio web turnos.laplata.gov.ar. En la cual se puede observar cada problema de accesibilidad detectado junto a la cantidad de veces que se han detectado cada tipo de smell.

Problema	Cantidad
Controles de formulario sin etiquetar.	7
Imágenes sin atributo ALT	11
Using h1-h6 to identify headings	1
Presencia de listas vacías	5
Enlaces consecutivos de texto e imagen al mismo recurso	9
Inexistencia de elemento h1	2
Formulario sin método estándar de envío	2
Encabezados vacíos, sin texto en su interior	5
Enlaces sin contenido	207
Etiquetas de campos de formulario (LABEL) ocultas visualmente	4
Declaración de idioma del documento	2
Página 'bien formada'	132
Marcos sin título	3

Tabla 2.2.2: Resultado total de la ejecución de TAW en varios sitios web.

Como podemos observar en la Tabla 2.2.2, todos los accessibility smells reportados son problemas que se detectan analizando estáticamente el código de la página web.

OAW-Analizador WEB, Observatorio de Accesibilidad Web

El Observatorio de Accesibilidad es un instrumento para evaluar la accesibilidad de una muestra de sitios web de la República de Ecuador de acuerdo con las recomendaciones de las Pautas de Accesibilidad para el Contenido Web 2.0 (WCAG 2.0) [OAWEC].

La validación puede ser por código o URL. Al igual que TAW, se pueden obtener un reporte de bad accessibility smells que puede contener un sitio web. Este validador es usado por el Observatorio de accesibilidad para hacer la monitorización y evaluación periódica de la accesibilidad de los portales del sector público. Solo está disponible online para el personal de la Administración Pública, a través del Servicio de Diagnóstico en línea de la Comunidad Accesibilidad, pero en 2020 se liberó y se puede instalar localmente.

Ejecutamos la herramienta en el sitio de Turnos de La Plata, Figura 2.2.4 y el resultado fue el siguiente:

Reglas no cumplidas	Cantidad
Uso elementos label para asociar etiquetas de texto con controles de formulario	8
Uso atributos ALT en imágenes	3
Combinando imágenes y textos adyacentes en un enlace para un mismo recurso: B) Verificar qué enlaces de elementos adyacentes no redireccionen al mismo lugar	1
Separando la información y la estructura de la presentación para permitir diferentes presentaciones	4
Organizar una página usando encabezados	2
Uso unidades EM para tamaños de fuente	154
Combinando imágenes y textos adyacentes en un enlace para un mismo recurso: B) Verificar qué enlaces de elementos adyacentes no redireccionen al mismo lugar	1
Utilización del atributo language en el documento HTML	1
Proporcionar botones de envió	1

Tabla 2.2.3: Resultado del reporte generado por OAW.

Se probó la herramienta con las mismas aplicaciones web validadas con TAW, y los resultaron fueron que las reglas incumplidas más comunes fueron:

- Uso de atributos ALT en imágenes. (encontrados 12)
- Uso de elementos LABEL para asociar etiquetas de texto con controles de formulario. (encontrados 14)
- Proporcionando contenido en todos los elementos. (encontrados 19)
- Separando la información y la estructura de la presentación para permitir diferentes presentaciones. (encontrados >200)
- Uso de unidades EM para tamaños de fuente. Tener el tamaño de fuente del texto en medidas dinámicas logrará que el usuario pueda cambiar el tamaño de la fuente sin perder contenido. (encontrados >400)

Al igual que en los casos de Level Access (análisis automatizado) y TAW, en OAW los accessibility smells también son encontrados por análisis estático del código de la página web analizada.

AChecker- Web Accessibility Checker

AChecker es otro ejemplo de plataforma gratuita que ofrece una gran libertad al usuario para escoger el tipo de análisis a realizar, ya sea de una web, un archivo HTML o un fragmento de código, así como el conjunto de directrices de accesibilidad a aplicar durante el análisis [AChecker], como se puede ver en la siguiente figura, Figura 2.2.5.

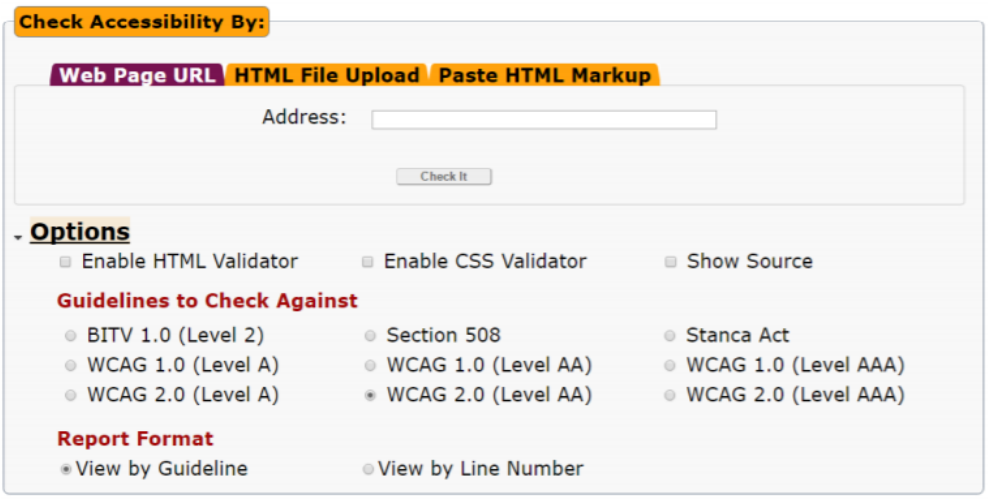


Figura 2.2.5: Captura que muestra las opciones del análisis con AChecker.

En el apartado de resultados nos encontramos con una forma de presentarlos similar a lo visto en TAW y OAW: una lista de los errores encontrados y problemas potenciales, así como su localización dentro del código fuente de la página, como se ilustra en la Figura 2.2.6. Otra opción interesante es la capacidad de descargar los resultados en varios formatos, y, además, escoger qué parte del análisis exportar. En la Figura 2.2.7 se muestra un ejemplo de análisis exportado a formato PDF.

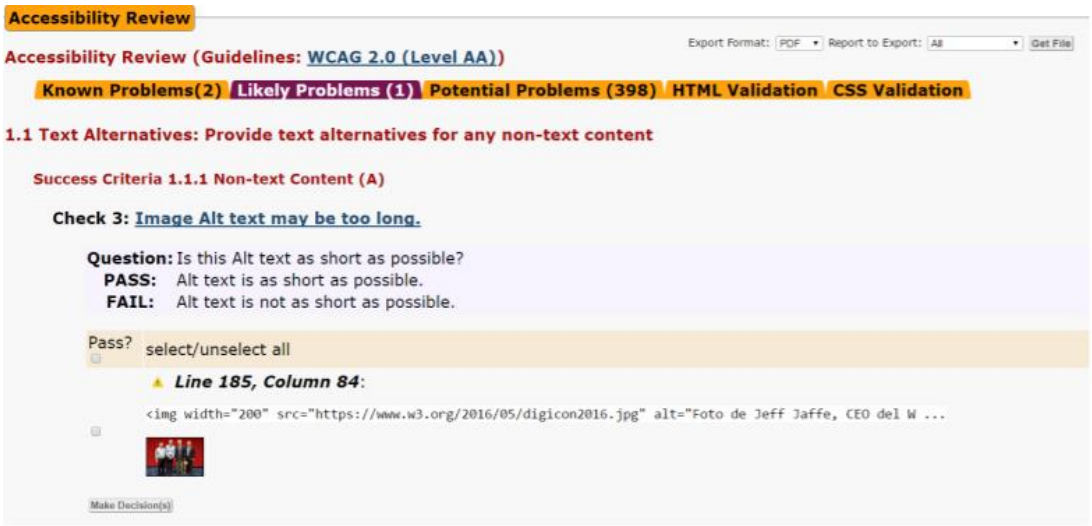
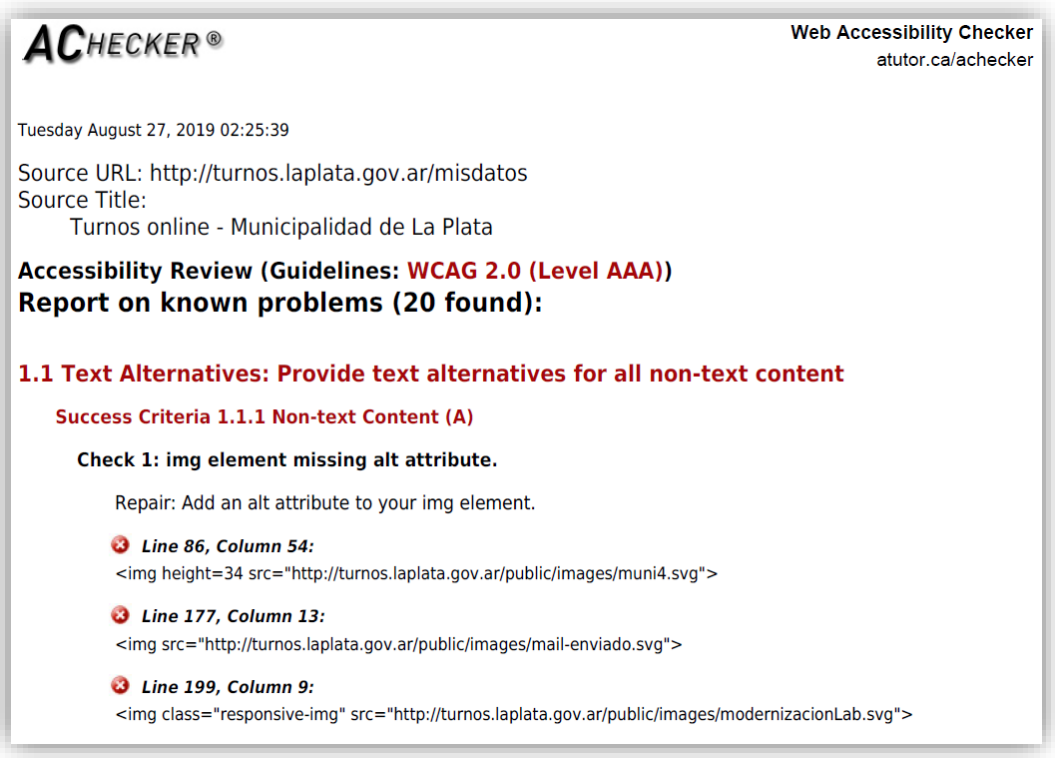


Figura 2.2.6: Resultado del análisis con AChecker.


ACHECKER®

Web Accessibility Checker
atutor.ca/achecker

Tuesday August 27, 2019 02:25:39

Source URL: http://turnos.laplata.gov.ar/misdatos
Source Title:
Turnos online - Municipalidad de La Plata

Accessibility Review (Guidelines: WCAG 2.0 (Level AAA))
Report on known problems (20 found):

1.1 Text Alternatives: Provide text alternatives for all non-text content

Success Criteria 1.1.1 Non-text Content (A)

Check 1: img element missing alt attribute.

Repair: Add an alt attribute to your img element.

Line 86, Column 54:
<img height=34 src="http://turnos.laplata.gov.ar/public/images/muni4.svg">

Line 177, Column 13:
<img src="http://turnos.laplata.gov.ar/public/images/mail-enviado.svg">

Line 199, Column 9:
<img class="responsive-img" src="http://turnos.laplata.gov.ar/public/images/modernizacionLab.svg">

Figura 2.2.7: Resultado en formato PDF con AChecker.

Los problemas encontrados más comunes usando AChecker en las mismas páginas webs de los testeos anteriores los vemos en la Tabla 2.2.4.

Accessibility smell	Turnos La Plata	Sitio Web City Bell	ABSA Tramites comerciales	CABA Tramites	Total
Imágenes sin atributo ALT.	**3**	**30**	**9**	**1**	**43**
Element input, de tipo "text", sin un label asociado, el label no posee texto.	**4**	**4**	**-**	**2**	**10**
b (bold) element usado, i (italic) element usado.	**2**	**1**	**34**	**2**	**39**
El contraste entre el color del texto y su fondo no es el suficiente para cumplir WCAG2.0 Level AAA.	**2**	**21**	**1**	**-**	**24**
Header nesting – el titulo cumpliendo h1,h2 es incorrecto.	**1**	**-**	**1**	**1**	**3**
No se encuentra el lenguaje del documento.	**1**	**-**	**1**	**-**	**2**
Atributo ID no es único.	**1**	**-**	**1**	**-**	**2**
Elemento link <a> sin texto.	**-**	**-**	**-**	**1**	**1**

Tabla 2.2.4: Resultado con los smells detectados más comunes utilizando AChecker.

2.2.3 Conclusión de herramientas de búsqueda estática

Como hemos podido ver, existen diversas páginas que cumplen con el cometido de analizar la accesibilidad web, y cada una presenta unas características y funcionalidades propias. Aunque al revisar los reportes generados, vemos que todos los smells detectados surgen en base al análisis estático del código y el catálogo es siempre el mismo. Todas las herramientas analizadas detectaron en su mayoría los mismos problemas.

2.3 Trabajos relacionados

2.3.1 Refactorings

Los refactorings fueron concebidos originalmente como una técnica para mejorar la calidad interna del software -como la comprensibilidad y mantenibilidad- mientras se preserva la semántica y funcionalidad [Fowler99]. Los refactorings solucionan problemas que se denominan “bad smells”. Asimismo, la técnica de refactoring también se propuso para resolver los problemas de accesibilidad y usabilidad para personas con discapacidad [Medina10] [Garrido13Pers]. Los refactorings de accesibilidad solucionan problemas que denominamos anteriormente en la sección 2.2 como “accessibility smells”.

Los refactorings no cambian el comportamiento de la app, sino que su propósito es mostrarles el camino a los usuarios para acceder al contenido de la app o comenzar su ejecución.

Desde el punto de vista del mantenimiento de una app WEB, un refactoring que es aplicado para incrementar su calidad externa tiene que apuntar a dos aspectos diferentes pero interrelacionados:

1. Permitir acceso a la información que antes era inaccesible (tranformation on accessibility).
2. Mejorar la interacción con el contenido que previamente se podía acceder, pero con algunas dificultades (transformation on usability).

Al mismo tiempo, una mejora (accesibilidad o usabilidad) puede beneficiar:

A. A un grupo de usuarios con características muy específicas que limitan su interacción con la aplicación.

B. A una audiencia mucho más ancha y general.

Claramente la última es la más deseable pero la más difícil de conseguir.

2.3.2 Client Side Web Refactoring

Para esta problemática, como describimos anteriormente, se creó la herramienta, Client-Side Web Refactoring (CSWR), que permite la creación automática de diferentes vistas personalizadas de la aplicación para resolver los bad smells que cada usuario reconoce [Garrido13Pers].

Un Web refactoring cambia la estructura de la navegación, visual o sentido de una aplicación web, preservando su contenido y operaciones, mientras se eliminan los bad smells de usabilidad o accesibilidad. El motor detrás de los CSWRs, usa un framework adaptativo del lado del cliente que apunta a adaptar las aplicaciones existentes cambiando la estructura del DOM de los mismos.

El enfoque de CSWR tiene dos beneficios clave:

- Mantenimiento simple, los desarrolladores mantienen una única app, aplicándole Web usability refactorings que se dirigen a un público en general, mientras que otras versiones diferentes de refactorings puede ser creadas para uno o varios usuarios diferentes.
- Independiente de la arquitectura.

El mayor inconveniente que tiene esta solución es que no existe una manera de reconocer automáticamente donde aplicar los refactorings, es decir los bad smells de accesibilidad que sufre un sitio web determinado. Hoy por hoy resulta muy difícil para una persona discapacitada reconocer qué refactoring instalar para solucionar los problemas que sufre.

Los refactorings permiten aumentar la usabilidad de una forma incremental usando el feedback de los usuarios, incluso en aplicaciones ya desplegadas en producción. Cada uno de estos refactorings está catalogado dependiendo el problema o “smell” que soluciona, en el caso de usabilidad, los llamamos **usability smells**.

2.3.4 Usability Smells Finder

En la herramienta denominada USF (Usability Smells Finder) se desarrolló una posible solución a la identificación de problemas de usabilidad, en la cual, el objetivo es proporcionar una detección automática de usability smells sobre la interacción del usuario en aplicaciones web que ya están corriendo en producción. Esta estrategia ésta basada en el análisis de los eventos de interacción del usuario (UI Events) [Grigera17].

Así, se extiende el trabajo realizado con CSWR, con la unión especifica entre los eventos UI a los usability smells, definiendo nuevos usability smells y reportando dichos smells a un nivel abstracto lo cual permita que sea posible sugerir soluciones concretas para ellos en términos de refactorings.

Este proceso automático de USF se define en tres pasos:

1) Events Logging
2) Usability Smells Detection
3) Reporting

En Events Logging: el componente del lado del cliente, mina eventos de grano fino para filtrar y agregar aquellos que pertenezcan al nivel medio de eventos llamados usability events.

En la etapa de Usability Smells Detection, el componente del lado del servidor clasifica y analiza los eventos de usabilidad usando algoritmos especializados en descubrir usability smells.

Finalmente, el server-side component es responsable del Reporting step, donde se muestran los usability smells con sus respectivos refactorings sugeridos para resolver dichos refactorings.

USF permite encontrar bad smells de usabilidad, esta herramienta funciona como un servicio que reconoce “usability smells” a partir de los eventos de interacción. De esta manera, USF se diferencia de las herramientas que chequean los WCAG porque estas revisan problemas en el HTML resultante, mientras que USF observa los problemas que ocurren dinámicamente con la página. Los eventos que USF reconoce son mayormente **eventos de mouse**.

2.3.5 Kobold

Kobold es una herramienta que funciona como un servicio web y tiene dos funcionalidades importantes: por un lado, la detección automática de problemas de usabilidad (que a partir de la jerga de refactoring se denominan “usability smells”) a partir de logs de **eventos de interacción**, y por otro lado permite aplicar CSWR como soluciones a cada uno de los bad smells encontrados [Grigera17]. Las soluciones se generan automáticamente, aunque a veces es necesaria la intervención de un usuario para ingresar algunos valores requeridos para la generación de los refactorings.

La herramienta ofrece “Usabilidad como Servicio”, caso específico de Software como Servicio” o SaaS (SaaS – Software as a Service). A medida que se utiliza el sistema se escanean las interacciones de los usuarios en las aplicaciones buscando problemas de usabilidad. Cuando alguna dificultad es detectada y se supera el valor definido para su umbral de tolerancia, se genera un reporte acompañado de recomendaciones en formas de refactorings que se pueden aplicar, en la Figura 2.3.1 se puede observar una imagen descriptiva de la arquitectura de Kobold.

El acceso al sistema requiere que los usuarios se registren en una cuenta y generen un código snippet que debe ser incorporado en la aplicación objeto del análisis.

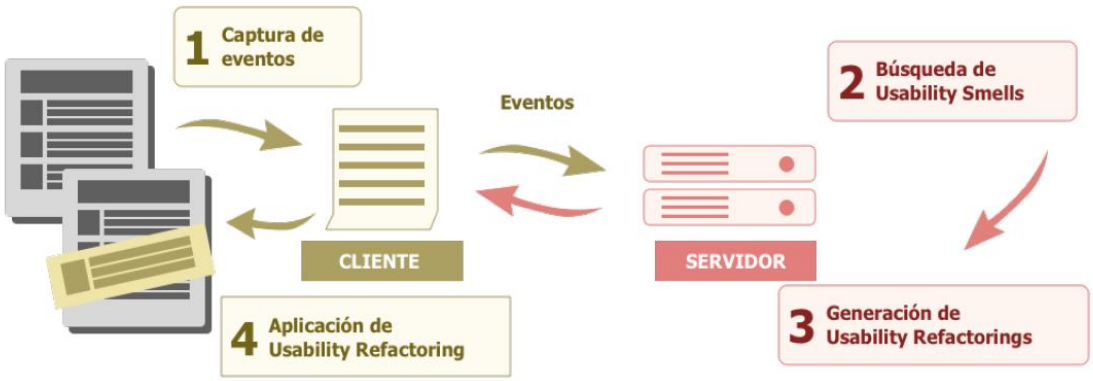


Figura 2.3.1: Arquitectura de Kobold, imagen obtenida de Self-Refactoring [Grigera18SR].

En el proceso intervienen dos elementos principales, un componente del lado del cliente y otro del lado del servidor. El primero se incrusta en la aplicación para capturar los eventos de interacción y aplicar las refactorizaciones sobre la interfaz. El segundo, procesa los eventos capturados por cliente, detecta los usability smells y reporta sugerencias de refactorings algunas de la cuales pueden aplicarse automáticamente. Todo este proceso está representado en la figura 2.3.2.

La desventaja de esta aplicación es que los reportes deben ser leídos por personas con experiencia técnica y existen algunos reportes que no pueden ser solucionados debido a que aún no existen refactorings desarrollados para este fin.

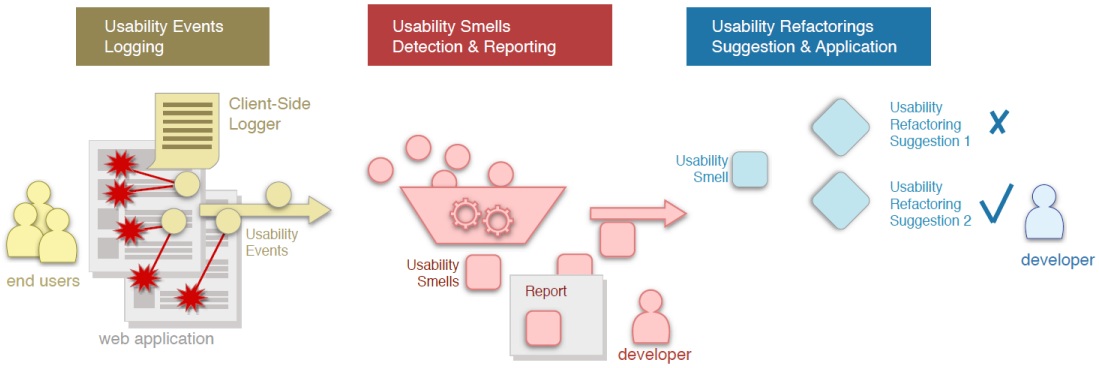


Figura 2.3.2: Proceso completo del approach automatizado implementado por Kobold. Imagen obtenida de Self-Refactoring [Grigera18SR].

2.3.6 Kobold accesible

Como se describió en la sección anterior, la estrategia utilizada en Kobold consiste en combinar componentes basados en el cliente, dentro del navegador web, con componentes en un servidor donde se realiza un análisis de tres etapas: Captura de Eventos, Detección de Usability Smells y Refactoring.

La captura ocurre en el lado del cliente, mientras que el análisis, se realiza en el servidor. Es el cliente quien evalúa las interacciones de los usuarios, filtrándolas y agrupándolas en eventos más abstractos llamados **eventos de usabilidad**. Mientras que, del lado del servidor, se clasifican y analizan esos eventos de usabilidad para descubrir problemas de usabilidad como **usability smells**.

Por el lado de la accesibilidad, se desarrolló como una extensión a Kobold, una versión accesible del mismo. A esta versión se la denominó **Kobold Accesible** [Durgam2020] en la cual mediante la captura de eventos de interacción (cliente) y análisis de eventos de accesibilidad (servidor) se realiza la búsqueda de patrones que permitan detectar dificultades y sugerir posibles soluciones. La arquitectura de esta nueva versión accesible la podemos ver en la Figura 2.3.3 a continuación.

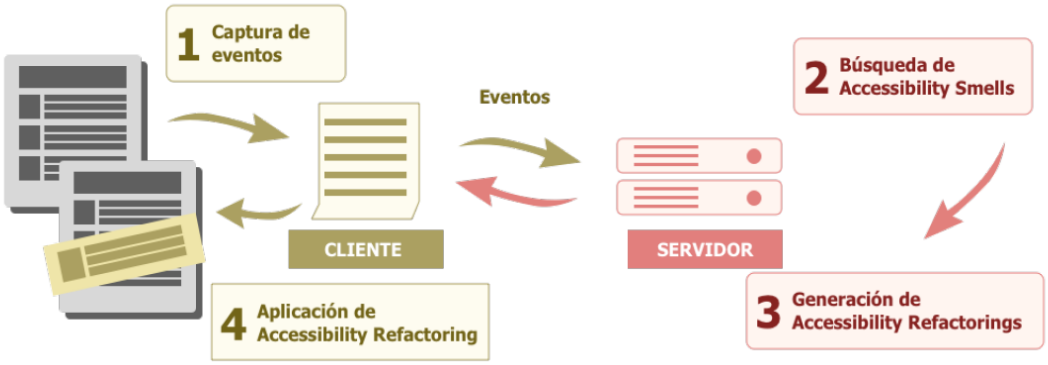


Figura 2.3.3: Arquitectura extendida de Kobold para los procesos para accesibilidad. Imagen obtenida de Detección de problemas de accesibilidad [Durgam2020].

A estos problemas de accesibilidad se los denomina **Accessibility Smells** y sus posibles soluciones **Accessibility Refactoring**. Estos smells señalan deficiencias en la interfaz que pueden ser detectadas y corregidas por los desarrolladores.

Los eventos de accesibilidad, al igual que los eventos de usabilidad, se diseñaron en función de las barreras que se esperan detectar y basándose en la observación de Accessibility Smells presentes en casos reales. En el trabajo realizado en Kobold Accesible, se estudiaron las acciones forzadas que deben realiza los usuarios que utilizan el teclado con el lector de pantallas NVDA y la posibilidad de capturarlas a través de interacciones sobre la interfaz.

Lograr vincular un problema de accesibilidad a una solución en términos de refactoring genera que sea posible automatizar progresivamente el proceso, reduciendo los costos y plazos de desarrollo de aplicaciones, haciéndolas más accesibles y liberando a los desarrolladores de la necesidad de experiencia para la exigente tarea de evaluación de accesibilidad.

2.3.7 Identificación automática de Widgets

Los menús desplegables son implementaciones de widgets basados en JavaScript y CSS que frecuentemente representan el mecanismo principal de navegación en una aplicación web. Estos widgets se utilizan para mostrar subniveles de enlaces de navegación que se presentan solo cuando los usuarios colocan el mouse sobre el widget. Sin embargo, muchos de los widgets disponibles en la Web no implementan los requerimientos de las aplicaciones de Internet enriquecidas accesibles (ARIA) que abordan cómo hacer que estos widgets sean accesibles.

En lo que respecta a esta falta de cumplimiento de las reglas ARIA, el investigador William Watanabe ha investigado cómo se implementó ARIA en 32 Widgets con pestañas (Tabs) y 74 implementaciones de widget de menes desplegable en múltiples sitios web según la lista de los sitios web más visitados realizada por Alexa (asistente de vos de Amazon) [Watanabe16]. Aunque el ARIA es la especificación que se rige como una recomendación de la W3C y es parte de la especificación HTML5, los resultados mostraron que son pocos los widgets que se implementaron utilizando las recomendaciones de ARIA.

Debido a esto, Watanabe desarrollo una herramienta que identifica automáticamente a los Widgets Menú de tipo Drop-Down en una aplicación web [Watanabe16]. Una vez que los elementos están identificados son analizados para determinar si están implementados utilizando las recomendaciones de ARIA.

Esta herramienta está basada en la simulación de escenarios de **interacciones de usuario**, **Mutation Observes** y **cambios de visibilidad** en la interfaz del browser. Y está compuesta por los siguientes componentes, Figura 2.3.4:

- **Web Robot** que genera eventos de navegación para simular la interacción de un usuario.
- **Especificación ARIA** la cual está representada por un conjunto de reglas ARIA especificadas en un archivo de configuración XML.
- **Evaluador** que consiste de un parser para el DOM, un clasificador de elementos, tests de especificaciones ARIA y ontología ARIA.
- **Manejador de resultados** que produce un reporte sumario de la evaluación.

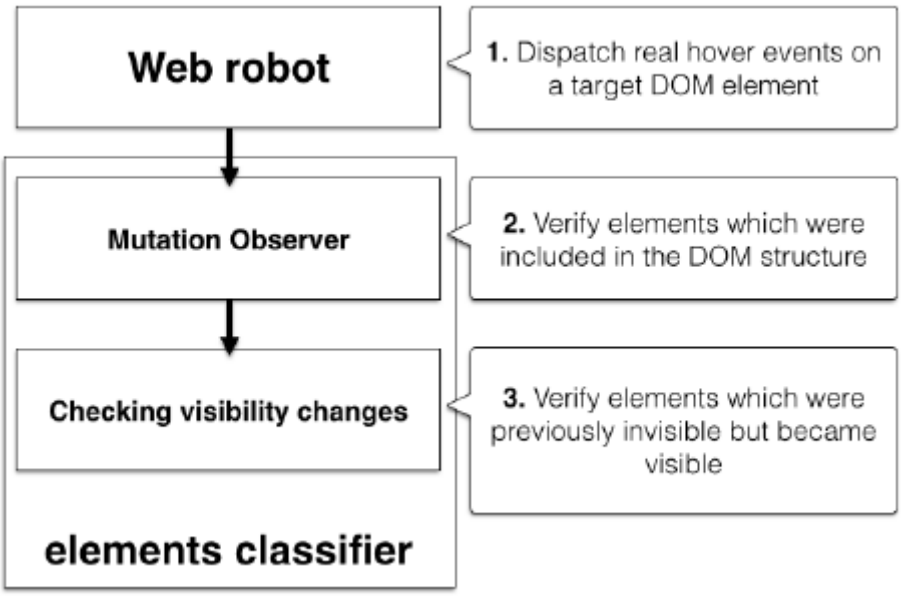


Figura 2.3.4: Esta figura muestra la relación entre cada uno de las fases del approach descripto en [Watanabe16].

Este approach simula a los usuarios interactuando con los menús desplegables, así se generan eventos de desplazamiento reales en los elementos que se muestran en una página web. Además, los cambios enviados al DOM son observados para clasificar elementos como menú desplegable o no, esta tarea es realizada por el componente clasificador de elementos. Esta detección se basa en analizar el resultado de la ejecución de código HTML, CSS y JavaScript dentro del navegador. Una vez que la herramienta concluye la detección del menú-desplegable, se analizan las reglas ARIA aplicadas a estos elementos y se generan los respectivos **reportes**.

2.3.8 Identificación de Menú drop-down usando clasificación de cambios de estructura HTML

Este es otro trabajo realizado basándose en el approach de Watanabe [Watanabe16], para lograr detectar widgets inaccesibles en páginas web [Antonelli2018]. La herramienta desarrollada agrega el uso de inteligencia artificial para poder clasificar automáticamente los cambios en el DOM, generados por los widgets de tipo menú drop-down.

El esquema de la herramienta, implica identificar los menús desplegables que se implementan mediante JavaScript. Los cambios en la estructura del DOM se capturan como eventos de tipo *mutation*, a través de la API de *MutationObserver*. Estos eventos contienen los cambios en los atributos que se van generando cuando el usuario interactúa con el menú desplegable.

Al igual que en el trabajo de Watanabe se utiliza un robot web para generar los eventos de interacción. Luego mediante los cambios de visibilidad, se clasifican los elementos de la página que pasan por estos cambios de visibilidad como también en sus atributos. También lo que se añade en este trabajo es una herramienta de tipo **web crawler**, el cual es un algoritmo que cumple la función de “bot” y se utiliza para analizar código de un sitio web es busca de informaciones, para después usarlas y clasificar los datos encontrados. Este crawler fue implementado usando Selenium Webdriver para el lenguaje Java, en donde se provee un ambiente de simulación para web browsers. En este ambiente el web robot va a generar interacciones con sitios webs como si fuera un usuario normal, mientras que el *crawler* va a ir recolectando diferente información al respecto. Como sabemos por el trabajo de Watanabe, este escenario de ejecución de la herramienta consume mucho tiempo para poder generar resultados.

Para este trabajo, Antonelli propuso utilizar el *web crawler* para tomar capturas iniciales de la estructura DOM de cada componente y elemento web de la página. También implementó **listeners JS** en los elementos de la página web a analizar para detectar cambios e interacciones por parte del usuario (en este caso el web robot). Luego al generarse los cambios en la estructura mientras el usuario interactúa se comparan estos cambios con los estados iniciales capturados por el *crawler*.

Varias de las técnicas de detección utilizadas por Antonelli fueron utilizadas para la implementación de alguno de los *finders* en esta tesina.

Una de las limitaciones que resalta Antonelli en su trabajo, es que la herramienta no puede detectar widgets implementados mediante código CSS, solamente puede analizar aquellos en los cuales hay código JS interviniendo en el widget. Menciona varios casos que encontró durante su investigación, donde detectó widgets que fueron implementados mediante el uso único de cambios CSS y no fueron analizados por la herramienta.

CAPÍTULO 3

ARQUITECTURA DEL PROYECTO Y TECNOLOGÍAS UTILIZADAS

3.1 Introducción

Para empezar este capítulo y con la explicación de la herramienta que se desarrolló para esa tesina, debemos dar un breve resumen de los términos de usabilidad y accesibilidad vistos en el capítulo anterior.

Podemos mencionar que la usabilidad se refiere a la facilidad de uso de nuestro producto digital. Por eso todos los contenidos y servicios de nuestra aplicación web deben ser fáciles de encontrar, navegar, leer e interactuar por los usuarios para alcanzar el objetivo de forma eficaz, eficiente y satisfactoria.

La accesibilidad aborda las experiencias de usuarios equivalente para las personas con discapacidad, incluidas las personas con discapacidades relacionadas con la edad y también usuarios inexpertos, con conexiones lentas y con dificultades con el idioma. Estas personas deben de percibir, comprender, navegar e interactuar con sitios web y otros productos digitales sin barreras.

Si nuestra aplicación web es difícil de usar, el usuario se pierde navegando y la información es difícil de leer y comprender, el usuario abandonará. Por eso nuestra aplicación web siempre debe indicar claramente lo que ofrecemos de una manera sencilla, clara e intuitiva, y con la mínima carga cognitiva para el usuario. Aunque lograr que esto sea concebido tiene un costo.

El costo asociado a la accesibilidad es el costo de la calidad y la prevención de errores. No existen costos directos asociados excepto los de la capacitación de los desarrolladores que deben incorporar nuevas formas de trabajar e incorporarlas a sus procesos habituales. Contar con técnicos no capacitados genera una pérdida considerable de tiempo en el proceso de evaluación por especialistas y re-trabajo para correcciones.

Si la accesibilidad no es un criterio más del producto y este proceso se realiza luego del desarrollo en horas de mantenimiento, el costo final es alto.

Por eso, la accesibilidad debe ser un requisito más para todos los nuevos desarrollos, y para los sitios y aplicaciones existentes se debe generar un plan de trabajo de adecuación progresivo garantizando que no se generen nuevas barreras de accesibilidad y las ya existentes se vayan adecuando progresivamente.

Para esto, no solo es necesario contar con técnicos capacitados, sino también existen herramientas automáticas de evaluación de accesibilidad. Estas ayudan a lograr cumplir con la meta de tener no solo una aplicación web usable sino también accesible.

En la sección 2.2.2, del Capítulo 2, describimos varias de estar herramientas automáticas. Sin embargo, tienen una gran desventaja, y es que, las búsquedas de problemas de accesibilidad son realizadas mediante análisis estático del código web de la página. Entonces el número de problemas de accesibilidad que pueden llegar a detectar es muy limitado. Logrando solamente cumplir con algunas de las reglas básicas de accesibilidad definidas por las WCAG [WCAG].

Existen varias herramientas diferentes que llevan al proceso de mejora de accesibilidad y usabilidad en aplicaciones web a un nivel superior, las cuales decidimos nombrarlas en esta tesina como *high-level usability and accessibility improvement tools*. Varios de estos desarrollos fueron definidos en la sección 2.3. Los cuales llevan la búsqueda y detección de smells a un nivel superior, detectando **accessibility** y **usability smells** no solo a nivel de búsqueda estática en el código, sino también dinámicamente a través de la observación de las interacciones del usuario, con es el caso de Kobold accesible [Durgam2020]. Estos eventos no pueden ser detectados durante una búsqueda estática.

Para continuar ampliando las opciones de herramientas para la mejora de accesibilidad web, en esta tesina se desarrolló una nueva propuesta la cual denominamos **Herramienta para la detección automática de problemas de accesibilidad**. La misma está enfocada en la idea de seguir con el desarrollo de nuevas herramientas para la búsqueda de *accessibility smells* que requieran búsquedas más complejas o dinámicas.

3.2 ¿Qué ofrece esta herramienta?

La herramienta para la detección automática de problemas de accesibilidad tiene como resultado final lograr generar reportes con *accessibility smells* presentes en un sitio web. En estos reportes se encuentran listados los diferentes *accessiblity smells* detectados por la herramienta junto a la información detallada de los elementos de la página que generaron estos problemas.

La herramienta está constituida por tres componentes principales:

1. **Extensión web**: la cual es la encargada de detectar elementos web que generen algún tipo de accessibility smell.
2. **API REST**: encargada de proveer la comunicación entre la extensión web, la base de datos y la aplicación de reportes.
3. **Aplicación de reportes**: es una UI en la cual tanto el propietario del sitio web como los desarrolladores pueden visualizar si su aplicación contiene algún *accessibility smell*.

Estos 3 componentes se pueden observar en la Figura 3.2.1.

En las siguientes secciones daremos más detalles de estos tres componentes principales.

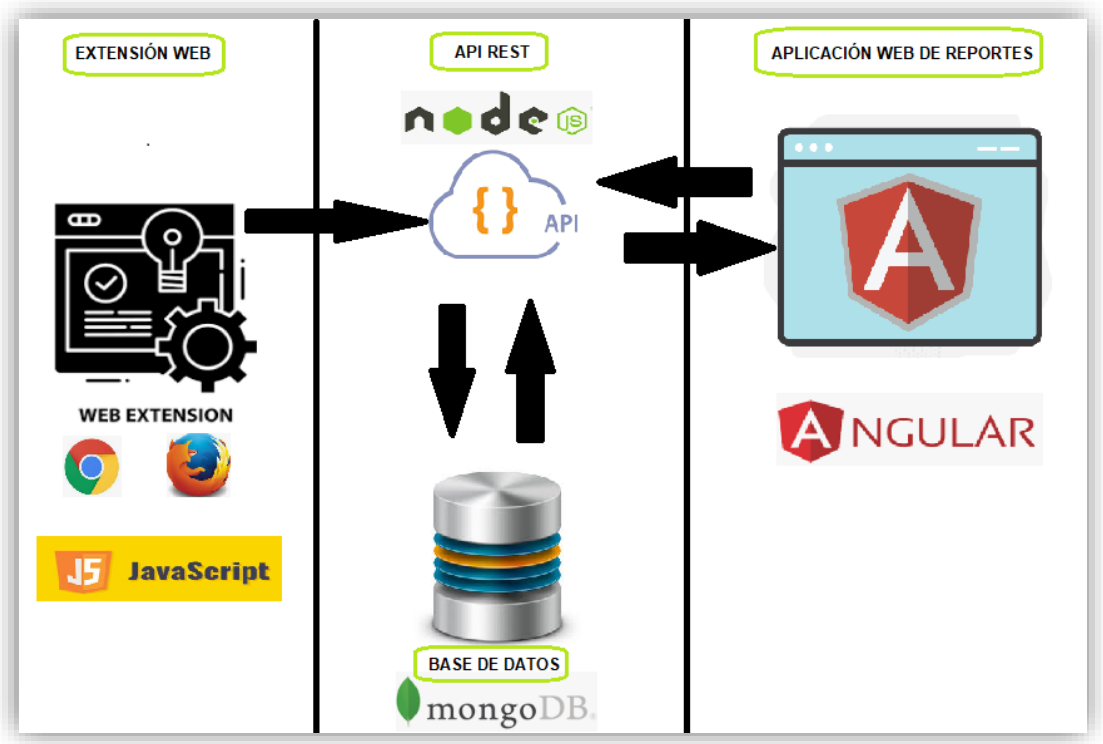


Figura 3.2.1: Diagrama general de la arquitectura de la herramienta desarrollada para esta tesina.

Detección de problemas y reporte

No importa la cantidad de smells presentes en la página, solo van a ser reportados aquellos que tengan su respectivo **finder** definido. Un finder es un algoritmo dentro de la herramienta, cuyo trabajo es detectar y reportar un smell especifico. Por ejemplo, en la Figura 3.2.2, se remarcaron dos conjuntos diferentes de elementos web que generan *accessibility smells* diferentes.

Por un lado, tenemos el caso de los botones inaccesibles mientras que, en el otro caso, los mensajes de error del formulario que también son inaccesibles.

Figura 3.2.2: Captura del sitio web de APR el cual contiene dos conjuntos de elementos diferentes que presentan problemas de accesibilidad.

Como veremos en capítulos siguientes, se implementaron diferentes finders dentro de la herramienta para lograr detectar diferentes tipos de problemas de accesibilidad y así generar un reporte de los problemas encontrados.

3.3 Diagrama de arquitectura

Las tecnologías utilizadas en el desarrollo de la herramienta utilizan un lenguaje uniforme, **JavaScript**, que proporciona dos mejoras importantes entre otras:

- en el rendimiento del software, ya que JavaScript tiende a ser muy rápido porque a menudo se ejecuta inmediatamente en el navegador.
- en la productividad de los desarrolladores. Por su **simplicidad**, La sintaxis de JavaScript está inspirada por Java y es relativamente sencillo de aprender comparado a otros lenguajes de programación populares como C++. Y su **popularidad**, JavaScript esta por todas partes de la web, y con la llegada de Node.js, se ha incrementado su uso en **backend**. Hay incontables recursos para aprender JavaScript. Tanto StackOverflow como GitHub muestran un creciente número de proyectos que usan JavaScript, y la popularidad que ha alcanzado en los recientes años se espera que siga creciendo.

Stack MEAN

Como mencionamos en la sección 3.2, la arquitectura de la herramienta está constituida por tres componentes principales, como se observa en el diagrama de la Figura 3.3.1. Si observamos este diagrama vemos que podemos clasificar a esta arquitectura como el tan popular **stack MEAN**. Este stack para desarrollo web basado en JavaScript es el fruto de la proliferación que ha llevado a JavaScript a todas las capas de desarrollo, empezando por el lado cliente en sus inicios (el navegador), pero yendo también al servidor y a la capa de almacenamiento. *En cualquiera de esos puntos podemos encontrar JavaScript listo para ser utilizado*.

Gracias a eso, hoy en día es posible crear aplicaciones distribuidas utilizando el mismo lenguaje JavaScript en todas sus fases y capas.

En la Figura 3.3.1 podemos observar el diagrama de cómo está compuesto una arquitectura basada en este stack. El cual podemos resumir de la siguiente manera:

- M = MongoDB/Mongoose.js: una base de datos muy popular actualmente y un elegante ODM (Object Document Mapper) para poder manipularla. Los registros almacenados en MongoDB son llamados documentos.
- E = Express.js: un framework ligero para aplicaciones web.
- A = Angular: un framework robusto para crear aplicaciones webs en HTML5 y JavaScript.
- N = Node.js: un intérprete server-side para JavaScript.

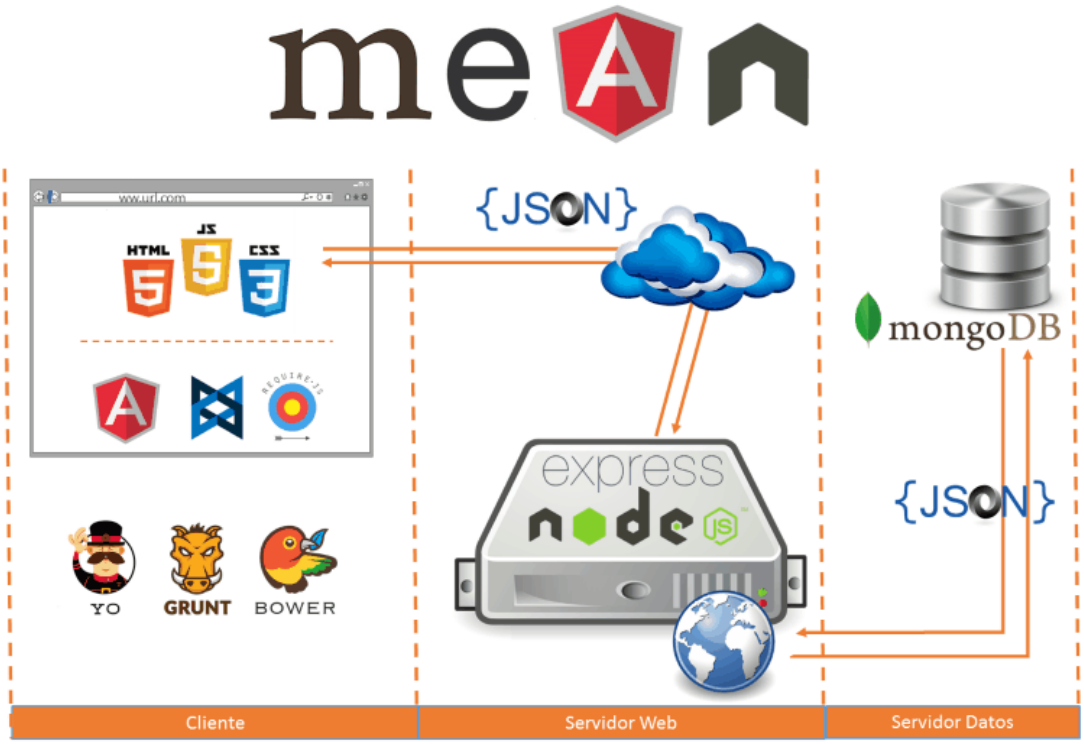


Figura 3.3.1: Diagrama de cómo está compuesto la estructura del stack MEAN.

División de la arquitectura de la herramienta

Previamente, en la sección 3.2, se mencionó los componentes que integran la arquitectura de la herramienta de detección, pero también la podemos describir en tres divisiones:

1. Detección.
2. Almacenamiento y manejo de la información.
3. Reporte.

Las cuales se describirán más en detalle en las siguientes secciones.

3.3.1 Detección

La instancia de detección podríamos decir que es la más importante de la herramienta. Ya que es donde se analiza y se detectan los problemas de accesibilidad en páginas web.

Como se logra observar en la Figura 3.3.2, usuarios que tienen instalada la extensión web de nuestra herramienta en sus web browsers, van a estar enviando información de sobre los smells detectados por los finders de la web extension mientras navegan por los diferentes sitios web.

Los **finders** son algoritmos implementados en la *web extensión*, que analizan el sitio web en busca de elementos que generen algún *accessibility smell*, donde varios de estos finder realizan **búsquedas dinámicas**. Esto quiere decir que dependen de los eventos de interacción del usuario para poder detectar elementos inaccesibles. En los capítulos 4, 5 y 6 se describen los finders de cada uno de los problemas de accesibilidad que detecta esta herramienta.

Luego de que un elemento es detectado como generador de un *smell* es automáticamente reportado en un formato JSON (*JavaScript Object Notation*) a la API REST de la herramienta.

Se eligió el formato JSON, ya que es un estándar actual basado en texto plano para el intercambio de información, por lo que se usa en muchos sistemas que requieren mostrar o enviar información para ser interpretada por otros sistemas, la ventaja de JSON al ser un formato que es independiente de cualquier lenguaje de programación, es que los servicios que comparten información por éste método, no necesitan hablar el mismo idioma, es decir, el emisor puede ser Java y el receptor PHP, cada lenguaje tiene su propia librería para codificar y decodificar cadenas de JSON. En el caso de nuestra herramienta de detección, tenemos cuatro aplicaciones distintas desarrolladas en tecnologías diferentes:

La extensión web en *JavaScript*, la API REST en *NodeJS*, la base de datos en *MongoDB* y la aplicación de reportes en *Angular*.

Las cuatro tecnologías trabajan cómoda y sencillamente compartiendo información en formato JSON.

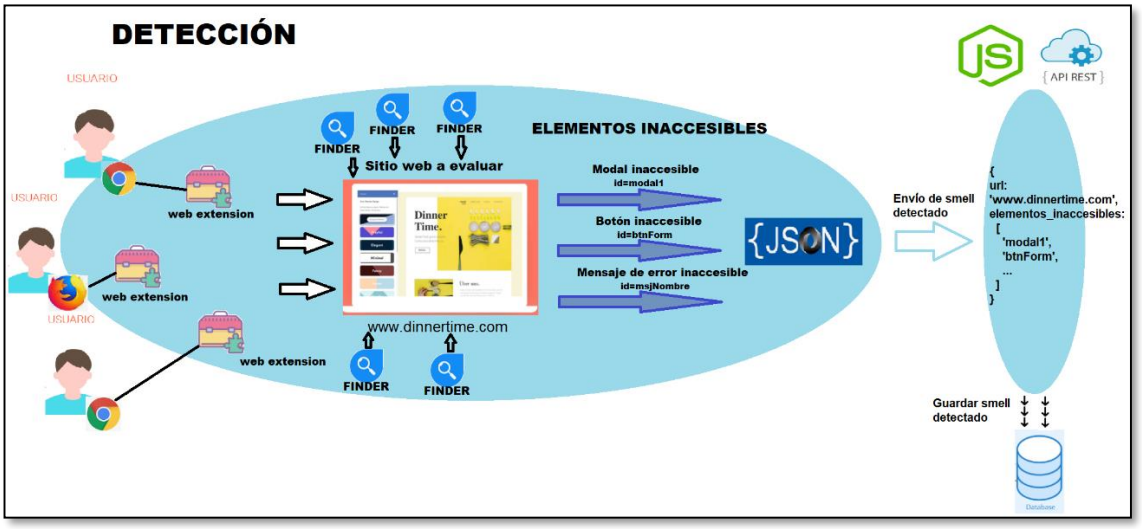


Figura 3.3.2: Diagrama del flujo de detección de la herramienta donde se logra observar a todos los actores involucrados.

Luego de que la información del elemento reportado es recibida en la API REST, esta es almacenada en la Base de Datos, para luego ser utilizada en los reportes de accesibilidad generados por la aplicación web de reportes.

3.3.2 Almacenamiento y manejo de la información (datos)

Almacenamiento de datos

Como se muestra en el diagrama de la Figura 3.3.3, la herramienta de detección almacena todos los datos generados por la extensión web en una base de datos MongoDB. Donde esta interacción se realiza a través de una API REST.

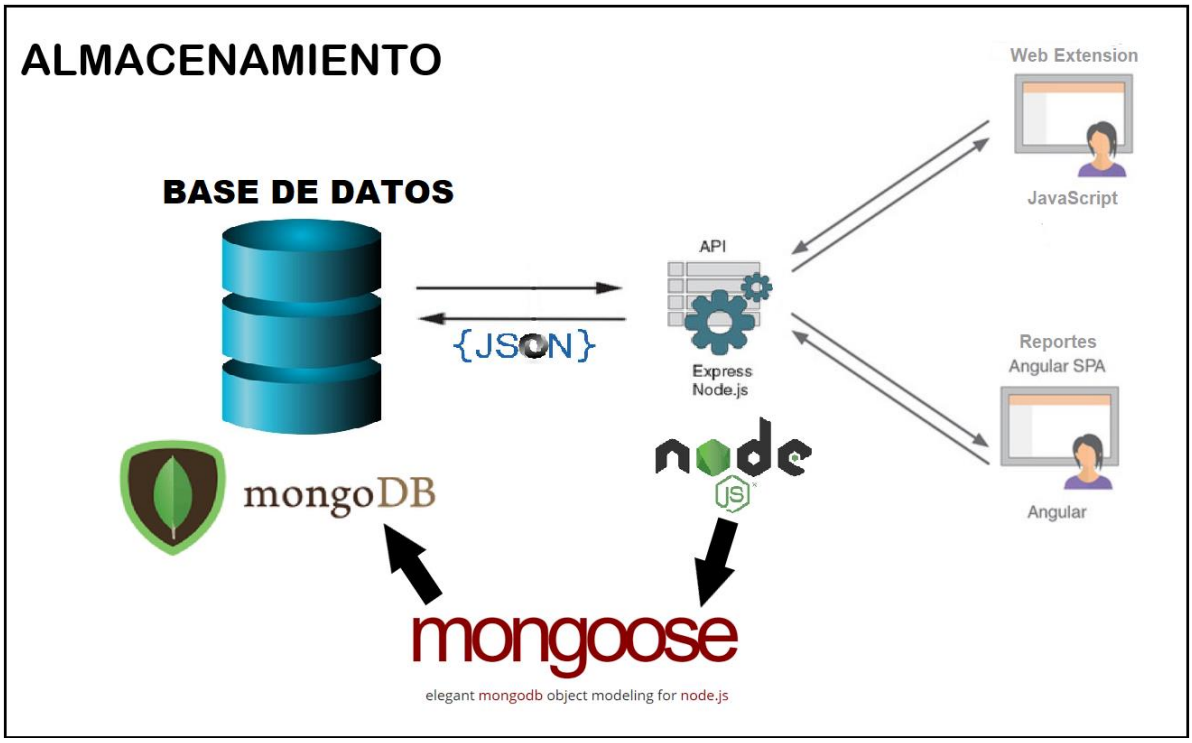


Figura 3.3.3: Diagrama del flujo de almacenamiento y manejo de datos en la herramienta de detección.

Manejo de la información a través de una API REST

Hemos utilizado el termino de API REST previamente sin dar una definición de su significado.

REST (Representational State Transfter), es un protocolo para el desarrollo de APIs. No es una arquitectura de software, sino un conjunto de restricciones que tener en cuenta en la arquitectura software que usaremos para crear aplicaciones web respetando HTTP.

REST utiliza un modelo cliente-servidor, donde el servidor es un servidor HTTP y el cliente envía peticiones HTTP (GET, POST, PUT, DELETE), junto con una URL y parámetros variables que están codificado. La URL describe el endpoint destino sobre el que actuar y el servidor responde con un código de resultado y un objeto JSON válido.

Cuando la API REST recibe una petición tanto para **obtener y almacenar datos** nuevos en la base de datos, la comunicación entre ambos se logra gracias a la librería llamada **Mongoose**. La cual nos permite definir los esquemas en nuestra API REST con la misma estructura que se está utilizando en los documentos de MongoDB.

Aun así, todo esto es transparente a los clientes (aplicaciones) que consumen esta información a través de la API REST, y es aquí donde radica la magia de estas APIs. No importa el nivel de complejidad ni la base de datos que manejen, la comunicación con ellas siempre es a través de peticiones HTTP y el resultado visible para quienes las consuman va a ser una respuesta JSON.

Node.js

Como mencionamos en secciones previas, la API REST está desarrollada en **Node.js**.

Node.js es un entorno de ejecución para aplicaciones interconectadas y del lado del servidor. Usa JavaScript y está disponible para muchas plataformas diferentes, como Linux, Microsoft Windows y Apple OS X.

Las aplicaciones de Node.js se crean utilizando muchos módulos de diferentes librerías y hay un ecosistema muy rico de librerías disponibles, algunas de las cuales se usaron para construir nuestra aplicación.

3.3.3 Reporte

Introducción

Para la parte de la visualización de los reportes generados por la herramienta, se desarrolló una SPA (Single Page Application) en Angular 10.

Se eligió esta tecnología por diferentes motivos además de ser un framework moderno.

Angular es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página (single page applications).

TypeScript es un lenguaje de programación libre y de código abierto. Es un superconjunto de JavaScript, que esencialmente añade **tipos estáticos** y **objetos basados en clases**. Esto permite que las aplicaciones sean más fáciles de mantener, cualquier cambio que deba hacerse en la aplicación podrá llevarse a cabo rápidamente y sin errores.

¿Cómo se generan los reportes?

La aplicación de reportes no tiene acceso directo a la base de datos, para ella la obtención de los datos es transparente. Ya que la API REST es la encargada de entregarle la información requerida para armar los reportes, como se muestra en la Figura 3.3.4. Esto es una gran ventaja ya que puede cambiarse la tecnología utilizada en la base de datos, por ejemplo, migrar de MongoDB a MySQL, y esto no va a requerir realizar ningún cambio en la aplicación de reportes.

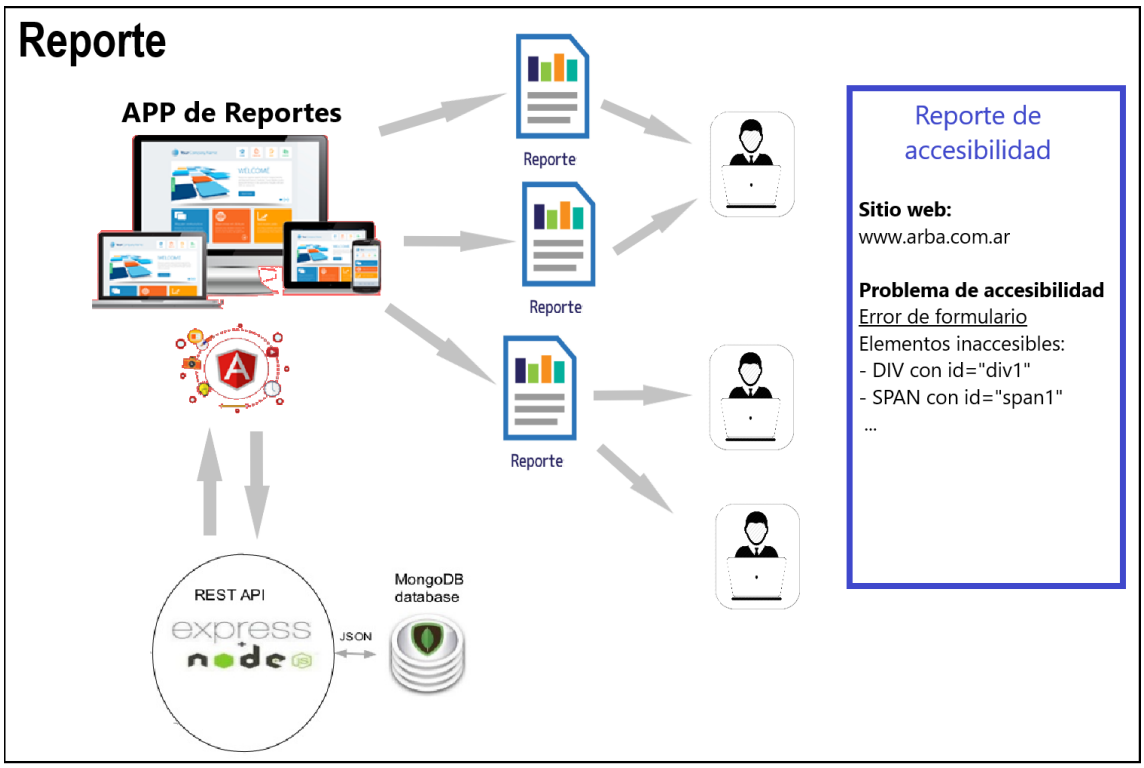


Figura 3.3.4: Diagrama del flujo de la generación de los reportes de accesibilidad.

Para esta tesina, los reportes van a poder ser accedidos por cualquier usuario. La aplicación contiene una sección principal donde están listadas todas las URL de páginas por donde se ejecutó la extensión web. Luego, al seleccionar cualquier de estas URLs se enviará a su respectivo reporte de problemas de accesibilidad, donde se muestran los tipos de *accessibility smells* y si se han detectado elementos que los provoquen. En los capítulos 4, 5 y 6 se muestran varios ejemplos de cómo son visualizados estos reportes.

Además, una vez en la página del reporte se puede exportar este mismo a un archivo PDF presionando el botón de “Descargar reporte como PDF”. En la Figura 3.3.5 se muestra un ejemplo de reporte como archivo PDF.

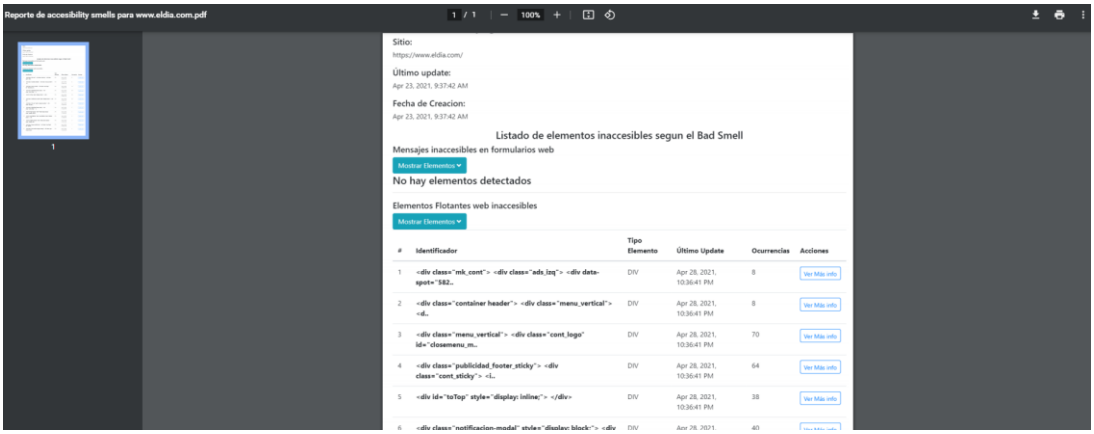


Figura 3.3.5: Ejemplo de reporte exportado como PDF.

3.4 Conceptos utilizados

En esta sección se describirán varios de los conceptos técnicos de programación, herramientas y patrones que se utilizaron a lo largo de esta tesina. Se puede decir que cada una de las siguientes subsecciones representan a conceptos fundamentales para implementar la herramienta de detección desarrollada para este trabajo. Por lo tanto, ameritan tener una sección aparte para dar una mejor definición y explicación sobre cada uno.

3.4.1 Observables

Introducción. Patrón Observer

Si bien la idea de esta tesina no es profundizar en el concepto de observables JavaScript, a continuación, se menciona una breve introducción para poder entender el concepto de *Observable* ya que es un término que se menciona varias veces durante el documento.

Un Observable es una colección "*lazy*" de valores a la cual se puede "subscribir" para así poder tener acceso a los valores. Observable es un nuevo sistema "*Push*" para JavaScript; este produce múltiples valores "pusheando" a los *Observer* (consumidores).

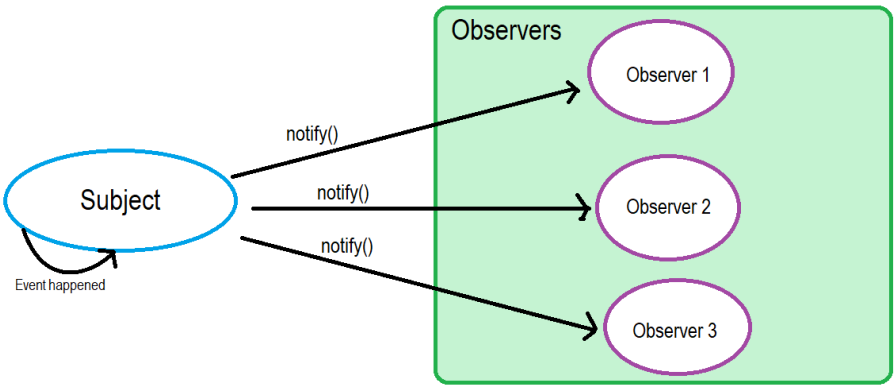


Figura 3.4.1: Grafico obtenido del sitio web [betterprogramming.pub], Patrón de diseño *Observer* [ObsVsPub-Sub].

Una de las soluciones propuestas en esta tesina surgió en base al patrón *Observer*, el cual tiene como propósito definir una dependencia de *uno-a-muchos* entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él [GAMMA+95].

¿Qué es un sistema "push"? Patrón Publisher-Subscriber

Antes de definir como es la utilización de un sistema push en JavaScript, debemos mencionar que esta idea surgió basándose en una variación al patrón *Observer* el cual se denominó como el patrón de diseño *Publisher-Subscriber* [Buschmann08], el cual ayuda a mantener el sincronizado el estado de los componentes cooperantes. Para lograr esto, permite la propagación *one-way* (unidireccional) de cambios: un *Publisher* notifica a cualquier cantidad de *subscribers* sobre los cambios en su estado. Esto significa que el *Publisher* envía estas notificaciones sin tener que conocer quiénes son sus observadores. Pueden suscribirse un número indeterminado de observadores para recibir notificaciones [Buschmann08].

Este patrón se aplica en cualquiera de las situaciones siguientes:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes son dichos objetos. En otras palabras, cuando no queremos que estos objetos estén fuertemente acoplados.

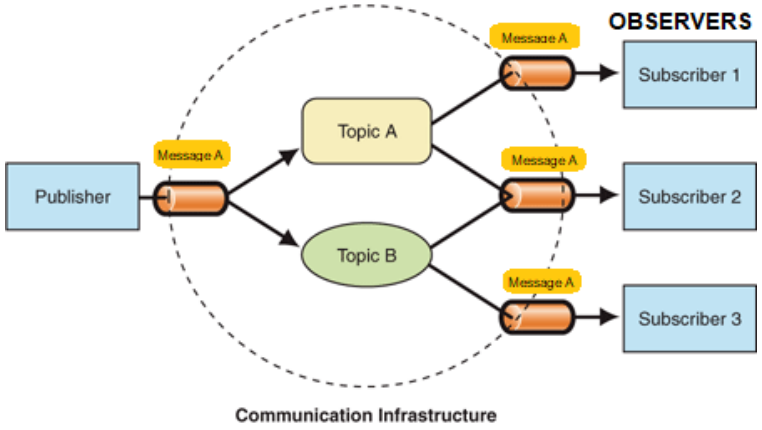


Figura 3.4.2: Gráfico representativo que describe como está compuesto el patrón *Publisher-Subscriber*.

Patrón Observer vs Publisher-Subscriber

Aunque estos patrones sean muy similares conceptualmente tienen sus diferencias entre ambos:

- En el patrón *observer*, los *observers* son conscientes del estado del *Subject*. El *Subject* mantiene un listado de los *Observers*. Mientras que, en *Publisher-Subscriber*, los *publishers* y *subscribers* no necesitan conocerse entre ellos. Simplemente se comunican con la ayuda de colas de mensajes o un *broker* (sistema de mensajería). Tal como se puede ver en la Figura 3.4.3.
- El en patrón *Publisher-Subscriber*, los componentes están débilmente acoplados a diferencia del patrón *observer*. Esto permite que los *publisher* y *subscribers* sean independientes entre sí. Facilita la interacción en cierto grado cuando sea necesario. Se puede distinguir fácilmente entre ambos lados.
- El patrón *observer* se implementa principalmente de forma sincrónica, es decir, el *subject* avisa a todos sus observadores cuando ocurre un evento. El patrón *publisher-subscriber* se implementa principalmente de forma asincrónica (utilizando una cola de mensajes).
- El patrón *observer* debe implementarse en un espacio de direcciones de una sola aplicación. Por otro lado, *publisher-subscrib*er es más un patrón *cross-aplication*.

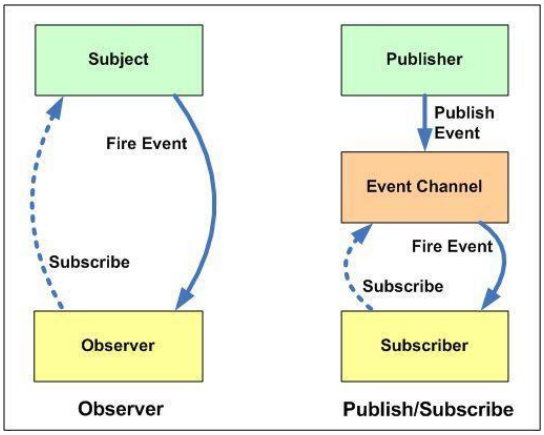


Figura 3.4.3: Grafico obtenido del sitio web [betterprogramming.pub], Diferencias entre el patrón *Observer* y *Publisher-Subscriber* [ObsVsPub-Sub].

Luego dar la definición sobre el patrón *Publisher-Subscriber* podemos continuar con la descripción de cómo se define este patrón en JavaScript. Un sistema *push* es donde el Producer determina cuándo enviar la información al *Consumer*. El *Consumer* no está al tanto de cuándo recibirá la información. En la figura 3.4.2 podemos ver cómo está representado este flujo.

Las *Promise* o promesas son el sistema de *Push* más común en JavaScript. Una promesa (Producer) entrega un valor resuelto a *callbacks* registrados (Consumer), pero, al contrario de las funciones, es la *Promise* la cual se encarga de determinar precisamente cuando el valor es "pusheado" a los *callbacks*.

Observable es como un *Promise*; con la diferencia que un *Promise* solo puede otorgar un solo valor, mientras que Observable puede otorgar múltiples valores [PromVsObs].

Anatomía de un Observable

Los *Observables* son creados usando el constructor o el operador de creación; son subscritos a un *Observer*, se ejecuta para entregar notificaciones ***next***, ***error*** y ***complete*** al *Observer* y su ejecución puede ser terminada.

Hay cuatro conceptos cuando tratamos con Observables:

- Creación de Observable
- Subscripción a Observable
- Ejecución del Observable
- Desechando el Observable

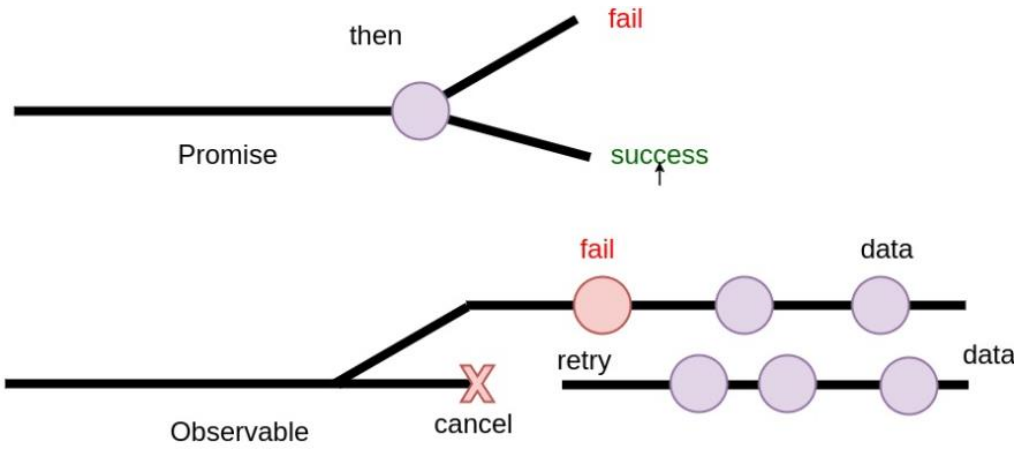


Figura 3.4.4: Grafico obtenido del sitio web [dev.to, Reactive Programming in JavaScript] que demuestra visualmente la diferencia entre una promesa y un observable en JS.

3.4.2 Mutation Observer

El Mutation Observer es un *API Web*, que nos permite detectar cambios en el *DOM* [MozillaWebAPIs].

Con esta API podremos saber cuándo se añade o elimina un elemento en el *DOM*, si se modifican atributos de los elementos, si hay cambios en el *textContent* de dichos nodos, o incluso si hay cambios en los hijos de los elementos.

MutationObserver es una nueva API de los navegadores modernos, y viene a reemplazar al antiguo método *MutationEvents*, el cual ya está obsoleto.

El método MutationObserver

Este constructor crea y retorna un nuevo *observer* el cual invoca una función de *callback*, cuando ocurre algún evento en el *DOM* (document object model).

Como usar MutationObserver

La implementación del *MutationObserver* en una aplicación es realmente fácil. Necesitamos crear una nueva instancia del *MutationObserver*, pasándole una función como *callback*, que se invocará cada vez que se produzca una mutación.

El primer argumento que nos devolverá, es una colección de todas las mutaciones, que hayan ocurrido.

Cada mutación proporciona información sobre su tipo y los cambios que se han producido.

Sintaxis:

```
const observer = new MutationObserver(callback);
```

callback

Es una función que será llamada cada vez que ocurra un cambio en el *DOM*.

Lógicamente, dichos cambios serán observados en el elemento o sub-elementos, que nosotros decidamos.

Un *MutationObserver* de tipo *object*, y será devuelto, cuando haya cambios en el *DOM*.

```
// configuration of the observer:
const observerOptions = { attributes: true, childList: true, characterData: true };
observer.observe(elementoHTML, observerOptions);
```

3.4.3 DOM API

Introducción

A continuación, se van a dar las definiciones de varios conceptos importantes referidos a la *DOM API* y su manipulación, los cuales se nombran a lo largo de esta tesina y son de suma importancia para el desarrollo de la extensión web.

Document Object Model o DOM ('Modelo de Objetos del Documento' o 'Modelo en Objetos para la Representación de Documentos') es esencialmente una interfaz de plataforma que proporciona un conjunto estándar de objetos para representar documentos *HTML*, *XHTML* y *XML*, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos *HTML* y *XML*, que es para lo que se diseñó principalmente [DOMWiki].

El responsable del *DOM* es el World Wide Web Consortium [W3C].

El *DOM* permite el acceso dinámico a través de la programación para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos con lenguajes como *ECMAScript* (JavaScript).

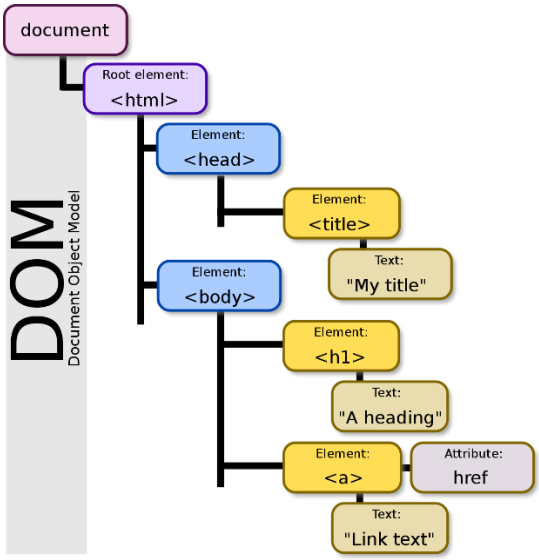


Figura 3.4.5: Grafico de ejemplo que muestra una estructura DOM clásica.

3.4.4 HTML DOM API

Definición

El *HTML DOM API* está compuesta por las interfaces que definen la funcionalidad de cada uno de los elementos dentro de un documento *HTML*, así como cualquier tipo de soporte e interfaces de los que dependan.

Las áreas funcionales incluidas en la *HTML DOM API* incluyen:

- Acceso y control de elementos *HTML* a través del *DOM*.
- Acceso y manipulación de datos de formularios.
- Arrastrar y soltar contenido en páginas web.
- Acceso al historial de navegación del navegador.
- Entre otros…

Estructura de un documento HTML

Como mencionamos anteriormente, el *DOM* es una arquitectura que describe la estructura de un *document*; cada documento está representado por una instancia de la interfaz *Document*. Un documento, a su vez, consta de un árbol jerárquico de nodos, en el que un nodo es un registro fundamental que representa un solo objeto dentro del documento (como un elemento o un nodo de texto).

Los nodos pueden ser estrictamente organizativos, proporcionando un medio para agrupar otros nodos juntos o para proporcionar un punto en el que se puede construir una jerarquía; otros nodos pueden representar componentes visibles de un documento. Cada nodo se basa en la interfaz *Node*, que proporciona propiedades para obtener información sobre el nodo, así como métodos para crear, eliminar y organizar nodos dentro del *DOM*.

Podríamos denominar a los nodos como recipientes vacíos. La noción fundamental de un nodo es que puede representar contenido visual que es introducida por la interfaz *Element*. Una instancia de un objeto *Element* representa un solo elemento en un documento creado usando HTML.

Por ejemplo, consideremos un documento con dos elementos (Figura 3.4.6), uno de los cuales tiene dos elementos más anidados en su interior:

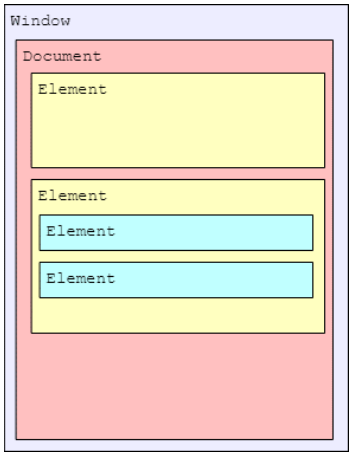


Figura 3.4.6: Grafico para demostración de ejemplo.

Si bien la interfaz *Document* se define como parte de la especificación *DOM*, la especificación HTML la mejora significativamente para agregar información específica para usar el *DOM* en el contexto de un navegador web, así como para representar documentos *HTML* específicamente.

Entre las cosas agregadas a *Document* por el estándar *HTML* se encuentran:

- Soporte para acceder a información diversa proporcionada por los encabezados *HTTP* al cargar la página, como la ubicación desde la cual se cargó el documento, cookies, fecha de modificación, sitio de referencia, etc.
- Acceso a listas de elementos en los elementos <head> y <body> del documento, así como listas de imágenes, enlaces, scripts, etc. contenidos en el documento.
- Soporte para interactuar con el usuario examinando el foco y ejecutando comandos en contenido editable.
- Entre otros…

La interfaz HTML element

La interfaz *Element* se ha adaptado aún más para representar elementos HTML específicamente mediante la introducción de la interfaz *HTML Element*, de la que heredan todas las clases de elementos HTML más específicas [MozillaWebAPIs]. Esto expande la clase *Element* para agregar características generales específicas de HTML a los nodos del elemento. Las propiedades agregadas por *HTML Element* incluyen, por ejemplo, *hidden* e *innerText*. *HTMLElement* también agrega todos los controladores de eventos globales para poder manipular el elemento.

Un documento HTML es un árbol DOM en el que cada uno de los nodos es un elemento HTML, representado por la interfaz *HTML Element*. La clase HTML *Element*, a su vez, implementa Node, por lo que cada elemento es también un nodo (pero no al revés). De esta manera, las características estructurales implementadas por la interfaz *Node* también están disponibles para los elementos HTML, lo que permite que se aniden entre sí, se creen y eliminen, se muevan, etc.

La interfaz HTML Element es genérico, sin embargo, proporciona la funcionalidad común a todos los elementos HTML como el ID del elemento, sus coordenadas, el código HTML que componen el elemento, la información sobre la posición de desplazamiento, y así sucesivamente.

3.4.5 ARIA

¿Qu es ARIA?

ARIA son las siglas de **Accessible Rich Internet Applications** (Aplicaciones Ricas Accesibles de Internet) y fue creado por el Consorcio de la World Wide Web (W3C). Define una forma de hacer que el contenido web y las aplicaciones web sean más accesibles para las personas con discapacidades. Ayuda especialmente con el contenido dinámico y controles avanzados de interfaz de usuario desarrollados con HTML, JavaScript y tecnologías relacionadas [WAIARIA].

Hoy en día, como mejor práctica, los desarrolladores web utilizan HTML semántico al construir sitios y páginas web. El HTML semántico es donde se utilizan los elementos HTML más apropiados y descriptivos, ayudando a los robots, a los servicios y a aquellos que dependen de las tecnologías de asistencia (AT) a comprender el significado del contenido.

ARIA va un paso más allá, agregando un conjunto de atributos que ayudan a definir el contenido de una página más allá de lo que el HTML puede hacer por sí solo. Estos atributos ARIA añaden cosas como navegación accesible, consejos sobre formularios, mensajes de error y otros útiles potenciadores.

En la Figura 3.4.7 hay un ejemplo de HTML semántico, hecho incorrecta y luego correctamente:

```
<body>
    <div>Welcome to my Website.</div>
    <span>This article is about ARIA in HTML.</span>
    <div>
        Lorem ipsum dolor sit amet consectetur,
    </div>
    <div>&copy; 2020</div>
</body>
```

Figura 3.4.7: Código HTML de ejemplo, para demostrar un mal desarrollo semántico.

Aquí, podemos ver un elemento *body* que contiene texto anidado usando etiquetas *div* o *span*. Aunque esta página puede ser estilizada para que se vea perfectamente bien usando CSS, sería difícil para las personas que confían en los lectores de pantalla entender completamente este contenido porque no está construido adecuadamente.

Buena semántica

```
<body>
    <header>
      <p>Welcome to my Website.</p>
    </header>
    <main>
        <h1>ARIA in HTML.</h1>
        <p>
            Lorem ipsum dolor sit amet consectetur, adipisicing elit.
        </p>
    </main>
    <footer>&copy; 2020</footer>
</body>
```

Figura 3.4.8: Código HTML de ejemplo, para demostrar un buen desarrollo semántico.

Como observamos en la Figura 3.4.8, en este ejemplo hemos hecho el mismo contenido más accesible utilizando los elementos HTML correctos en la estructura de la página.

1. El *header* contiene el contenido introductorio.
2. El elemento *p* sirve como logotipo.

3. *main* indica que el usuario está entrando en el contenido principal de la página.
4. Un elemento *h1* indica el título del contenido.
5. Otra etiqueta *p* indica un párrafo.
6. Un *footer* indica que el usuario ha llegado al final de la página; contiene cosas como información sobre derechos de autor y enlaces adicionales.

Con el HTML semántico es más fácil para los usuarios que confían en los lectores de pantalla para entender el contenido de la página; los lectores de pantalla pueden interpretar los elementos de HTML semántico bien utilizados.

Otro ejemplo común que involucra el uso de mala semántica involucra el uso de *divs* como botones en HTML, estilizándolos para que se vean como un botón real con CSS.

¿Qu son los roles de ARIA?

Siempre que sea posible, se debe utilizar el HTML semántico nativo, pero se pueden utilizar los roles ARIA para rellenar cualquier hueco. Los roles ARIA son aplicados a elementos HTML usando el atributo *role*, y pueden ser usados:

- para describir los elementos más nuevos o conceptuales que podrían no tener un soporte completo para el navegador o ser entendidos por los lectores de pantalla, por ejemplo *<button role="tab">Tab</button>.*
- para "arreglar" (lo mejor posible) el HTML implementado incorrectamente donde no se ha usado la semántica, por ejemplo * Button *.

Con estos ejemplos, los usuarios no videntes, con la ayuda de lectores de pantalla, podrán identificar los elementos como pretendemos.

Clasificación de los roles de ARIA

Los roles de ARIA pueden clasificarse en 4 tipos;

1. **Roles abstractos:** De acuerdo con la documentación oficial del W3C, estos son los roles utilizados por el navegador. Son la base sobre la que se construyen todos los demás roles de WAI-ARIA. Los autores del contenido no deben utilizar roles abstractos porque no se implementan en la vinculación del API. Ejemplos de roles abstractos incluyen *widget*, *landmark*, *window*, *command*, etc.
2. **Widget Roles:** Los roles de los widgets se utilizan para definir un elemento de interfaz de usuario cuando no se utiliza el HTML semántico. Según el W3C, algunos roles de widget sirven como interfaces de usuario independientes o actúan como parte de un widget compuesto más grande. Algunos ejemplos de roles de los widgets son *alert*, *dialog*, *checkbox*, *marquee*, etc. Otros roles de los widgets sirven como contenedores que gestionan otros contenidos de widgets. Ejemplos de estos widgets son *tree*, *tablist*, *menu*, *menubar*, etc.
3. **Estructura del documento:** Se trata de funciones que describen las estructuras que organizan el contenido de una página y que no suelen ser interactivas. Roles como estos ayudan a las tecnologías de asistencia a identificar y clasificar las diferentes secciones de una página. Algunos ejemplos son *article*, *toolbar*, *row, list*, etc.

4. **Roles de referencia:** Roles de referencia que identifican grandes áreas de contenido. Esto ayuda a las tecnologías de asistencia a definir estas secciones a los usuarios que dependen de ellas. Ejemplos de funciones de referencia son *application*, *form*, *main*, etc.

Llevando los papeles aún más lejos, también podemos añadir estados y propiedades de ARIA a los elementos.

¿Qu son los estados y propiedades de ARIA?

Los estados y propiedades de ARIA proporcionan apoyo a los roles de ARIA que existen en una página. Cuando se combinan con los roles, pueden suministrar tecnologías de asistencia con información adicional de la interfaz de usuario. Siempre que hay cambios en los estados o propiedades, las tecnologías de asistencia son notificadas de este cambio para que puedan alertar al usuario de que se ha producido un cambio.

Las propiedades de ARIA se diferencian de los estados en que el valor de una propiedad (como *aria-labelledby*) suele ser menos probable que cambie a lo largo del ciclo de vida de la aplicación, mientras que es probable que un estado (por ejemplo, *aria-checked*) cambie con bastante frecuencia debido a la interacción del usuario. Las propiedades y estados de ARIA pueden clasificarse como:

1. Atributos del Widget
2. Atributos de las regiones vivas
3. Atributos de arrastrar y soltar
4. Atributos de la relación

Atributos del Widget

Estos tipos de atributos se utilizan para los elementos comunes de la interfaz de usuario que reciben las aportaciones del usuario y procesan las acciones del usuario. También se utilizan para apoyar las funciones de los widgets. Entre los ejemplos figuran los siguientes: *aria-readonly, aria-required, aria-checked*.

Atributos de las regiones vivas

Estos atributos se utilizan para indicar a un usuario que un contenido específico puede cambiar o actualizarse (o, simplemente, que se trata de regiones vivas). Se aplican directamente sobre el elemento que cambia para que las tecnologías de asistencia puedan notificar a los usuarios el cambio en dicho documento/página. Por ejemplo, sería importante hacer uso de *aria-live* en un elemento que muestra datos en vivo desde un punto final como la bolsa de valores. Ejemplos de atributos como este son *aria-live*, *aria-atomic*, *aria-busy* y *aria-relevant*.

Atributos de arrastrar y soltar

Se utilizan para elementos que son arrastrables o son objetivos de lanzamiento. Las formas que requieren la carga de cualquier tipo de archivo (imagen, documento pdf, etc.) a menudo utilizan el método de arrastrar y soltar para cargar dicho archivo. Es importante que no se deje de lado a los usuarios que dependen de tecnologías de asistencia, y este tipo de atributo puede ayudar a ello. Ejemplos de ello son el *aria-dropeffect* para los elementos que reciben el **evento de soltar** y el *aria-draggable* para los elementos que son arrastrables.

Atributos de relación

Estos tipos de atributos ayudan a conectar dos elementos que son difíciles de identificar de otra manera utilizando la estructura del documento HTML. Un ejemplo de este tipo de atributo es el *aria-labelledby* por atributo. Este atributo ayuda a conectar un campo de entrada con una etiqueta en un formulario (pero puede utilizarse para relaciones menos obvias), en la Figura 3.4.9 vemos un claro ejemplo del uso de este atributo.

```
1. <form>
2.     <label for="name" id="nameLabel">Name</label>
3.     <input type="text" aria-labelledby="nameLabel">
4. </form>
```

Figura 3.4.9: Código HTML de ejemplo, mostrando el uso del *aria-labelledby*.

Otros ejemplos de atributos de relación son *aria-colspan, aria-controls, aria-owns, aria-flowto*, etc.

Reglas de uso de ARIA

Cuando se utiliza ARIA, hay algunas cosas que se deben y no se deben hacer para asegurarse que un sitio web o el proyecto en el que se está trabajando sea accesible a los usuarios que dependen de lectores de pantalla y otras tecnologías de asistencia. Estas son conocidas como las Reglas de Uso de ARIA y son cinco, pero en este caso vamos a nombrar dos de ellas ya que son las vamos a hacer referencia en varios problemas de accesibilidad a lo largo de esta tesina.

ARIA Regla #1

*Si puedes usar un elemento o atributo HTML nativo con la semántica y el comportamiento que requieres **ya construido**, en lugar de volver a utilizar un elemento y añadir un rol, estado o propiedad ARIA para hacerlo accesible, **entonces hazlo...***

Esto significa que es mejor usar HTML semántico donde esté disponible tanto como sea posible y no recurrir a usar elementos como *div* en lugar del elemento *button* y agregar roles, estados o propiedades ARIA para hacer que tales elementos sean accesibles. Hay algunos casos en los que esto podría no ser posible e incluyen:

1. La función no está disponible actualmente en HTML
2. Si las restricciones del diseño visual descartan el uso de un elemento nativo en particular porque el elemento no puede ser estilizado como se requiere.
3. Cuando el elemento HTML nativo no tiene soporte de accesibilidad.

ARIA Regla #2

Esta regla de uso de ARIA establece que todos los elementos interactivos deben tener *nombres accesibles*, es decir, el nombre de un elemento de la interfaz de usuario. Tomemos este ejemplo:

```
<label>Email</label>

<input type="email" />
```

Esto es incorrecto porque las tecnologías de asistencia no podrían conectar el valor de la etiqueta con el campo *input* y sus usuarios también tendrían dificultades para comprender que el campo está destinado a su dirección de correo electrónico. En su lugar, se puede hacer cualquiera de las siguientes implementaciones;

```
<label for="email">Email</label>
<input type="email" name="email" id="email">
```

O

```
<label> Email <input type="email"> </label>
```

3.4.6 Tecnología de Extensiones de navegador

3.4.6.1 Web Extensions

Las extensiones web o de navegador, **web extensions**, son pequeños programas que pueden modificar y mejorar la funcionalidad del navegador, permitiendo así a las aplicaciones web adaptarse a los requerimientos del usuario, por ejemplo, Google Chrome, Mozilla Firefox, Microsoft Edge entre otros. Estas se desarrollan utilizando tecnologías web como es HTML, JavaScript y CSS.

Principales características de las web extensions:

- Tienen acceso a gran parte de los componentes y el comportamiento del navegador.
- Las modificaciones necesarias se hacen mediante manipulaciones del DOM, del lado del cliente.
- Diferentes navegadores implementas una misma API.

Las extensiones web tienen poca o ninguna interfaz de usuario. Por ejemplo, la imagen a continuación en la Figura 3.4.10, muestra un ícono a la derecha del browser que al hacer click en él se abre una pequeña interfaz.

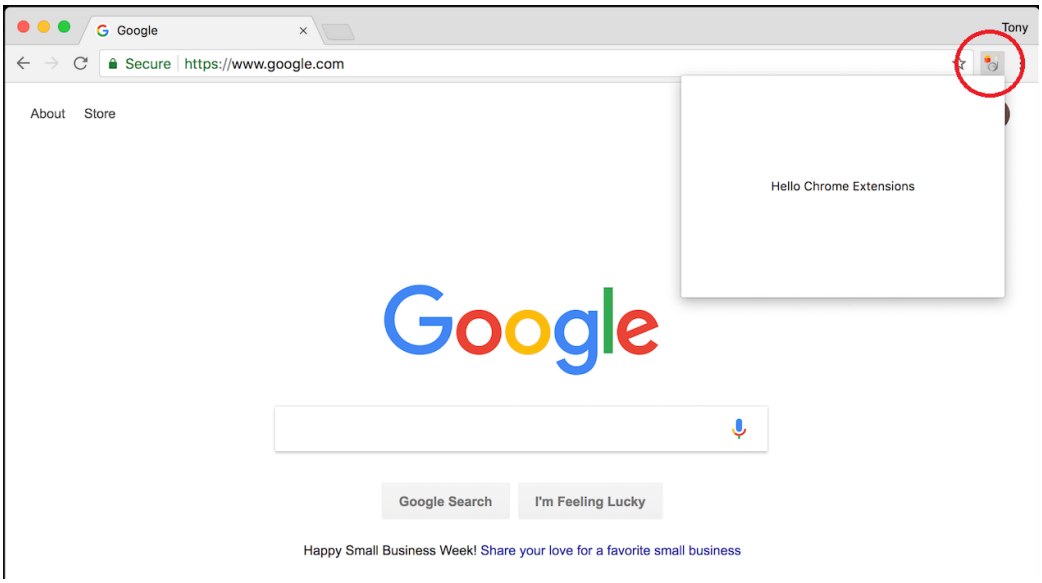


Figura 3.4.10: Ejemplo de extensión web con interfaz simple.

Las extensiones son archivos que se encuentran empaquetados en uno solo, los cuales el usuario descarga e instala. Este empaquetado, a diferencia de las aplicaciones web normales, no necesita depender de contenido web.

Tal como fue expuesto anteriormente, las extensiones permiten añadir funcionalidad al web browser sin sumergirse profundamente en código nativo. Se pueden crear nuevas extensiones con tecnologías básicas con las que mayormente trabajan los programadores en desarrollo web: HTML, CSS y JavaScript.

Las extensiones para Firefox (add-ons) están construidas utilizando la tecnología cross-browser de la **WebExtensions API** [MozillaWebAPIs]. Esta tecnología es compatible en gran medida con la extensión API soportada por browsers basados en **Chromium** (como Google Chrome, Microsoft Edge, Opera, Vivaldi) [ChromeExtensions]. En la mayoría de los casos, las extensiones escritas para browsers basados en Chromium corren perfectamente en Firefox con solo algunos cambios.

3.4.6.2 Configuración

El punto de entrada a la extensión es el archivo **manifest.json**. Este archivo, obligatorio, es necesario ubicarlo en la raíz de la estructura. Aquí es donde se declaran propiedades como el nombre de la extensión, su versión, las rutas de los iconos, los permisos a los que debe acceder, etc.

Aunque hay cierta estandarización en cuanto a las propiedades, cada navegador puede tener las suyas propias, o incluso, que alguna de ellas sea obligatoria, como ocurre en el caso de Microsoft Edge, el cual necesita que esté declarada la propiedad ***author*** para poder ejecutarla. Mientras que para Chrome las propiedades base requeridas son el ***name***, ***version*** y ***manifest_version***.

En la propiedad "***content_scripts"*** se indican los archivos que se ejecutarán dentro del contexto de la pestaña activa, y puede acceder a las APIS propias de cada navegador. Aquí indicaremos el contenido a inyectar en la pestaña activa que tengamos abierta, por ejemplo, el CSS que tenemos definido en la ruta de la propiedad ***"css"*** o los JS. Podemos afirmar que aquí se encuentra el *core* de la lógica de la web extension.

En este caso, el JS lo estamos inyectando por código, como veremos más adelante. Es interesante conocer todo lo que rodea a esta propiedad, para eso te recomiendo revisar la documentación que puedes encontrar en este apartado.

Destacamos la propiedad ***"default_popup"***, que es la que va a indicar el HTML a mostrar al clickear el icono de la extensión (se le podría llamar la sección *home*).

La propiedad ***"permissions***" es donde podemos indicar qué tipo de acceso tendrá la extensión sobre el navegador, la cual será informada al usuario a la hora de instalarla en el navegador.

Finalmente, ***"browser_action"*** básicamente indica cuál es el archivo a abrir cuando ejecutemos la extensión y la ruta de su icono a mostrar.

A continuación, en la Figura 3.4.11, podemos ver un ejemplo de archivo *manifiest.json*, usando las propiedades que hemos definido anteriormente. Con este contenido, ya tendríamos un archivo de manifiesto válido.

```
{
 "manifest_version": 2,
 "name": "Extension Name",
 "description": "Extension functionality ...",
 "version": "2.3",
 "icons": {
   "32": "images/icon_32.png",
   "128": "images/icon_128.png"
 },
 "permissions": [
   "",
   "file:///*/*"
 ],
 "browser_action": {
   "default_icon": "images/icon_32.png",
   "default_popup": "popup/popup.html"
 },
 "content_scripts": [
   {
     "matches": [ "http://*/*", "https://*/*", "file:///*/*"],
     "all_frames": true,
     "css": ["styles/tab/styles.css"]
   }
 ]
}
```

Figura 3.4.11: Captura del código de un archivo *manifest.json* de ejemplo.

3.4.6.3 Background Script

Cada extensión tiene una página background invisible que es ejecutada por el browser. Existen dos tipos:

- Persistent background pages
    - se mantiene activa todo el tiempo.
- Event pages
    - solo está activa cuando se necesita.

Google alienta a los desarrolladores a que utilicen event pages, porque esto logra reducir la utilización de memoria y mejora el rendimiento general del navegador. Sin embargo, también en bueno saber que aquí es donde se debe poner la lógica principal e inicialización de la extensión. Normalmente el **background page/script** juega el rol de puente conector entre las otras partes de la extensión.

En la Figura 3.4.12 se encuentra un ejemplo de cómo se debe describir en el *manifest*:

```
"background": {
    "scripts": ["background.js"],
    "persistent": false/true
}
```

Figura 3.4.12: Captura del código de la declaración del *background script*.

3.4.6.4 Content Script

Si se necesita acceso al DOM de la página actual, entonces se debe utilizar un **content script**. El código se ejecuta dentro del contexto de la página web actual, lo que significa que se ejecutará con cada actualización. Para añadir este script a la extensión web, se debe usar la sintaxis definida en la Figura 3.4.13.

```
"content_scripts": [
    {
        "matches": ["https://*/*", "https://*/*"],
        "js": ["content.js"]
    }
]
```

Figura 3.4.13: Captura del código de la declaración del *content script*.

Hay que tener en cuenta que el valor de ***"matches"*** determina para cuales páginas se va a utilizar el script.

Resumen

En la tabla 3.4.1, se encuentra un breve resumen sobre las diferencias más importantes entre content y background scripts para lograr comprender en que situaciones se usan cada uno. Mientras que en la Figura 3.4.14 se describe en un gráfico los principales componentes de una web extension.

Background scripts	Content scripts
Se ejecutan en el contexto del navegador	Se ejecutan en el contexto de una página Web
Una sola instancia por navegador	Una instancia por pagina
NO pueden manipular el DOM	Pueden manipular el DOM
No tienen tantas restricciones como los scripts convencionales, ej: pueden evitar la SOP (same-origin policy) Estos scripts pueden acceder a diferentes páginas webs de distintas pestañas del browser sin restricciones.	Tienen las restricciones de un page-script.
Browser-side debugging tools, Figura 2.4.7. Acceso a multiple information general del browser.	Page-side debugging tools, Figura 2.4.7. Acceso a diferente información relacionada a la pestaña actual.

Tabla 3.4.1 Tabla con las principales diferencias entre background y content scripts.

Los web browsers en general ofrecen a los desarrolladores diferentes *tools* de debugging para ayudarlos durante el desarrollo. Dependiendo el tipo de script se tiene acceso a estas herramientas como vemos en la Figura 3.4.15.

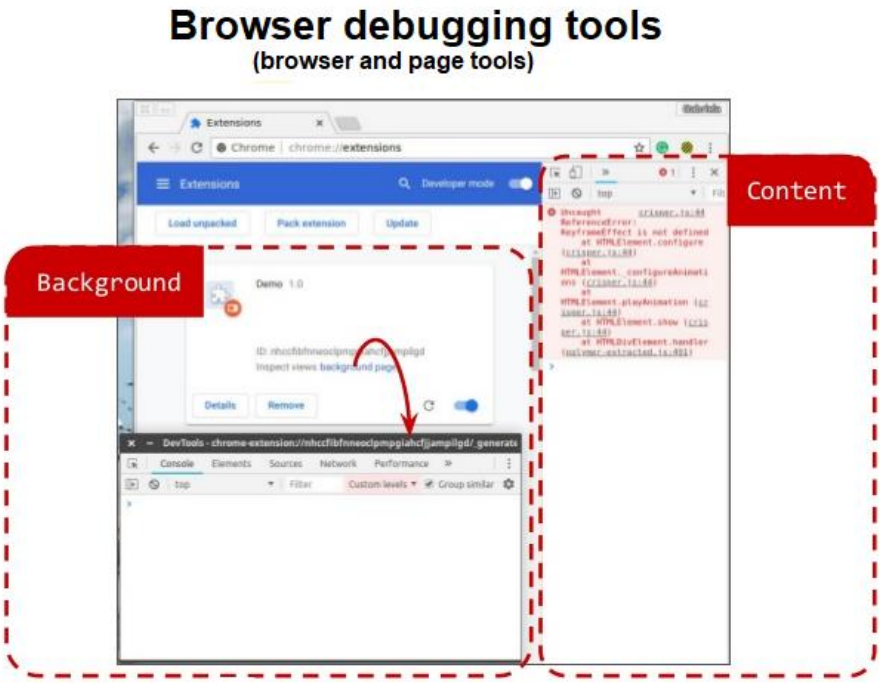


Figura 3.4.14: Tools para debugg presentes en el browser.

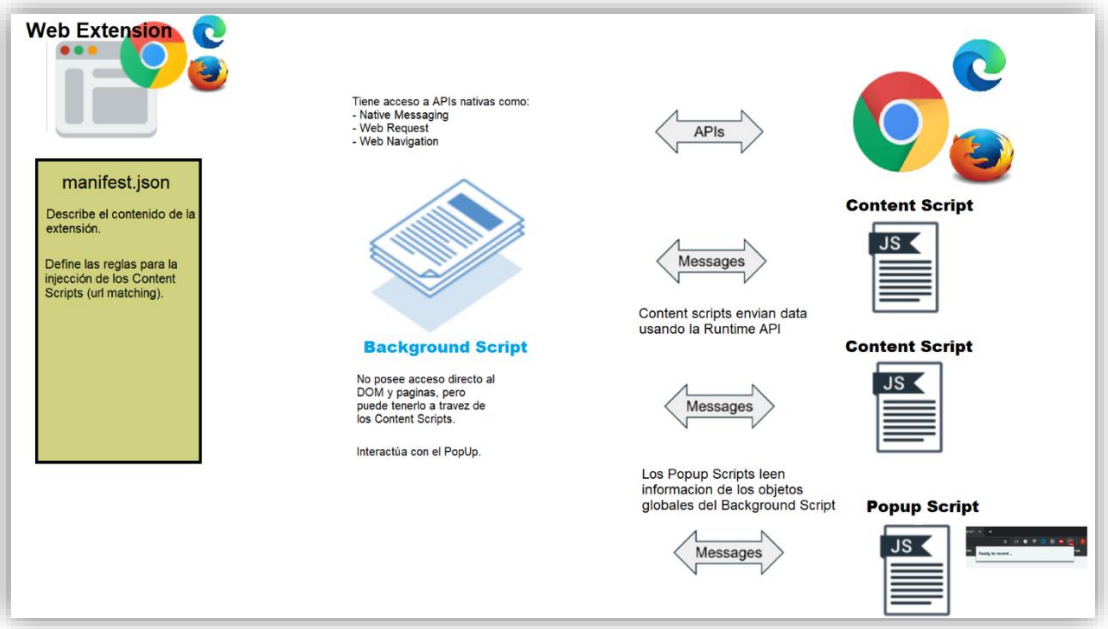


Figura 3.4.15: Grafico que resume los componentes principales de una web extensión y la interacción entre ellos.

3.5 Diferencias con trabajos relacionados

Como se ha mencionado anteriormente, la lógica de detección esta implementado en los **finders** para los diferentes problemas de accesibilidad, dentro de la **extensión web**.

La decisión para incorporar estos *finders* en una extensión web se tomó en base a dos factores importantes:

1. La necesidad de tener acceso la estructura de componentes web del DOM del sitio web a analizar. Este acceso sin restricciones al DOM es una de las características más destacables de las extensiones web, además de proveer de varias APIs web para analizar todos los tipos de elementos web junto a sus características.
2. Mediante el uso de **listeners JS** en los elementos que se necesitan analizar es posible capturar los eventos de interacción que realizan los usuarios al interactuar con dichos elementos. Tales como eventos de click, teclado, focus, hover y muchos más.

Es decir que a diferencia del trabajo desarrollado por Watanabe, no se simula la interacción de los usuarios, sino que se observa la interacción real, lo cual permite evitar los problemas encontrados por Watanabe [Watanabe16].:

- **Decremento de la precisión por la generación de falsos positivos y falsos negativos**

  Como describimos en la sección de trabajos relacionados, la herramienta de Watanabe mediante un simulador web trata de simular que genera un usuario al estar navegando por un sitio web. Y describió que el principal problema de los simuladores es que puede generar falsos positivos.

  En su herramienta se focalizo en los elementos desplegables de tipo drop-down menú, donde el 30% del resultado fueron falsos positivos. Esto quiere decir que la herramienta detectó como drop-down menú a elementos que no lo son. En este caso, los elementos de tipo tooltip tienen el mismo comportamiento que los drop-down menú. Watanabe no encontró la forma de que su herramienta lograra diferenciar entre ambos elementos.

  En la herramienta de detección que se desarrolló en esta tesina no fue el caso, ya que al tener acceso directo a los elementos del DOM y sus atributos ha sido posible desarrollar finders más precisos. Por lo tanto, se logró tener una detección más precisa de los elementos específicos que necesitan analizarse y así descartar falsos positivos. Esto se va a ver mejor reflejado en los capítulos dedicados a cada uno de los casos a analizar.

- **Decremento de la eficiencia por el tiempo requerido para la simulación**

  Watanabe menciona que la simulación lleva mucho tiempo y es muy ineficiente. En la herramienta desarrollada en su trabajo de investigación, el intervalo de tiempo necesario para analizar todos los elementos de una página web depende del número de elementos en la estructura del DOM.

  Donde las páginas analizadas en las pruebas de su herramienta tienen un promedio de 6 mil elementos y su análisis demora hasta 6 horas en promedio, lo cual es demasiado tiempo para realizar pruebas en tiempo real. Una solución propuesta es solo detectar los elementos que se activen mediante eventos de click, en esta categoría entrarían los drop-down menú, pero aun no encontraron como lograr esto.

  Este es otro problema que no sucede en nuestra herramienta ya que no necesita simular las interacciones del usuario, simplemente las captura logrando así una identificación de **accessibility smells en tiempo real**.

  Una vez que la extensión web está instalada en el navegador web, esta comienza a reportar mientras el usuario navega por diferentes sitios web.

Otra diferencia importante que caracteriza a nuestra herramienta y no menciona Watanabe en su trabajo es la de **extender la detección a otros tipos de elementos**. Con desarrollar el finder necesario para un grupo específico de elementos y agregarlo a la extensión web es suficiente. La única complejidad es la del desarrollo del finder, luego añadirlo a la extensión web es un trabajo sencillo.

CAPÍTULO 4

ANÁLISIS DE ACCESIBILIDAD EN ELEMENTOS WEB INTERACTIVOS CON HANDLERS JS ASIGNADOS

4.1 Introducción

Durante los primeros testeos usando NVDA para encontrar *accessibility smells* en diferentes sitios webs, uno de los primeros problemas hallados fueron los **elementos interactivos** que son ignorados por el *screen reader*. Esto quiere decir, durante estas búsquedas manuales, se observó que NVDA pasó por encima algunos elementos que tienen alguna funcionalidad dentro de la página sin mencionarlos. Es decir, que deberían poder ser accesibles, pero por algún motivo no lo son.

Para demostrar este tipo de problema, en la Figura 4.1.1 se muestra un ejemplo de elementos que parecen botones, tienen una funcionalidad asignada, pero NVDA los pasó por alto durante las pruebas.

Figura 4.1.1: Captura dentro del sitio web de APR, que contiene “botones” inaccesibles.

Los “botones” que se observan en la Figura 4.1.1 despliegan modales al presionar clic sobre ellos, pero utilizando teclado no hay forma de poner el foco sobre ellos y lograr accionarlos mediante la tecla *enter*. Revisando su código HTML, se observa que no están declarados con la etiqueta *<button>* como corresponde, sino que son elementos links, o sea elementos *<a>*. Por ejemplo, en el caso del botón “Ultima Boleta” posee la siguiente definición:

```
<a id="btn_ultima" class="registro">Última Boleta</a>
```

Aun así, NVDA debería poder leerlo, y como vemos en la definición del elemento, carece del atributo más importante de los elementos de tipo link: El atributo HREF, el cual define el destino de redirección de ese link. Entonces si no existe ese atributo, NVDA lo va a ignorar ya que parece un link que no hace nada.

Entonces surge la pregunta, dónde está definida la lógica para que estos elementos realicen alguna acción? En los archivos JS asociados al código de la página web. Estos scripts generalmente contienen código JavaScript que puede ser interpretado por el browser y así poder manipular el DOM de la página web.

Dicho esto, podemos observar en la Figura 4.1.2, la definición de dos funciones JS, cada una asociada a cada uno de los links previamente mencionados, mediante el ID del elemento. Estas funciones se quedan escuchando *eventos de click*, y cuando se realiza esta acción se ejecutan y muestran un modal de contenido.

```
$('#ul_co').click(function(){
    grecaptcha.reset();
    $("#btnpagarsl").show()
    $("#btnultimades").hide()
   $('#laventanaultboleta').dialog({modal:true,position:'center',resizable:false ,whidth:320});

});

$('#btn_ultima').click(function(){
    grecaptcha.reset();
    $("#btnpagarsl").hide()
    $("#btnultimades").show()
   $('#laventanaultboleta').dialog({modal:true,position:'center',resizable:false ,whidth:320});

});
```

Figura 4.1.2: Código JavaScript asociado a los botones de la Figura 4.1.1

Es así como se les da la funcionalidad a estos elementos links y junto a estilos CSS se los modela como botones, pero no lo son.

Este caso de elementos mal definidos que no son accesibles se repite en varios sitios web durante nuestra búsqueda de *accessibility smells*. Todos estos casos detectados siempre están asociados a funciones **handlers JavaScript**, y debido a la gran cantidad de casos encontrados, es que se decidió tratar de hallar la forma de poder detectarlos, analizarlos y reportarlos si es necesario.

4.2 Si necesitas un botón, usá el elemento <button>

Usando WAI ARIA junto con código JavaScript, es posible transformar un elemento para que se comporte exactamente igual que un botón (elemento <button>); esto significa que, en la mayoría de los escenarios posibles, tanto la interacción con el elemento como la información que recibe el usuario por parte del screen reader serán idénticos, de tal forma que serán completamente indistinguibles en la práctica, a menos que se analice el código.

Sin embargo, dado que el comportamiento del “disfrazado de botón” depende de otras tecnologías, esta complejidad añadida sin necesidad será más propensa a fallar si no se han contemplado todas las situaciones posibles.

Por ejemplo, si se usa un para enviar los datos de un formulario, como no se trata de un botón real necesitará de JavaScript para funcionar como tal, por lo que si éste no está disponible (desactivado desde el browser) el “botón” no funcionará en absoluto, mientras que un elemento <button> (o un <input>) de tipo submit funcionará incluso aunque JavaScript esté desactivado.

WAI ARIA no modifica la interacción

Además de la regla anterior, es muy importante tener claro desde el principio que el uso de WAI ARIA no cambia jamás la interacción original de los distintos elementos.

Por ejemplo, por defecto un no recibe el foco del teclado ni ejecuta acciones al hacer clic o pulsar una tecla. Por lo tanto, aunque se apliquen atributos para modificar la información semántica del de forma que se identifique como un botón, si no se añade la interacción mediante JavaScript se seguirá comportando igual, esto es, no recibirá el foco ni ejecutará acciones.

Para dejar más en claro los inconvenientes que estas prácticas generan, en la Figura 4.2.1 se definieron varios ejemplos simples.

```
// Un botón que recibe el foco
<button class="btn">Soy un button</button>
// Un DIV que no recibe el foco
<div class="btn">Soy un div</div>
// Sigue siendo un DIV que no recibe el foco
<div class="btn" tabindex="0">Soy un div</div>
// Rol button y que recibe el foco
<div class="btn" tabindex="0" role="button">Soy un div</div>
```

Figura 4.2.1: Código HTML de ejemplos ilustrativos en la definición de “botones”

Tomando en cuenta que el contenido de nuestra página sería solo los elementos definidos en la Figura 4.2.1; si comenzamos a navegar por el contenido mediante la tecla TAB, veremos que el primer elemento puede recibir el foco y el segundo no. La razón de esto, por supuesto, es que el primero es un elemento *button* y el segundo es un *div*. Sin embargo, se puede solucionar este problema agregando el atributo *tabindex=”0”* al elemento *div*, esto hace que un elemento que inicialmente no podía recibir el foco ahora lo pueda. Es por eso que el tercer y cuarto “botón” se pueden enfocar a pesar de que sean *div*s.

Por supuesto que el *div-button* puede ser enfocado, pero sigue comportándose como un *div* aun si se le agrega el *rol* de *button*. Ahora, qué sucede si les asignamos un *handler on click* a cada uno de los elementos de la Figura 4.2.1, si hacemos click sobre ellos se va a ejecutar ese handler JS. Aunque, si intentamos hacer lo mismo usando las teclas *Enter* o *Spacebar*, solo el primer botón va a disparar el evento para ejecutar el handler. Para que los *div-button* disparen esa acción hay que añadirles los event handlers para esas teclas.

Dicho todo esto, surge la pregunta: es necesario implementar tanta lógica para emular un botón utilizando un div? No lo es, es por eso que debemos utilizar el elemento <*button*> cuando necesitemos un botón.

Un problema de accesibilidad muy presente

Como mencionamos en la introducción de este capítulo, durante pruebas manuales usando NVDA, se han encontrado varios casos de elementos inaccesibles con *handlers JS* asignados. En el caso de los botones que están definidos utilizando *links*, *span's* y *div's*, se ha realizado una investigación muy interesante por parte del sitio web CSS-TRICKS en más de 8 millones de páginas web [CssTricks]. Y el resultado fue que hay una enorme cantidad de botones implementados sin utilizar el elemento <button>, como podemos observar en la Figura 4.2.2.

Element	Attribute & Value	Count
`a`	class=btn	3,251,114
`a`	class=button	2,776,660
`span`	class=button	292,168
`div`	class=button	278,996
`span`	class=btn	202,054
`div`	class=btn	131,950

Figura 4.2.2: Tabla obtenida de la investigación realizada por Css-Tricks [CssTricks].

Todos estos casos son potenciales elementos inaccesibles, debido a esto, en esta tesina se desarrolló un nuevo *finder* en la herramienta de detección, para detectar y reportar este tipo de accessibility smell.

4.3 Ejemplos de elementos inaccesibles con handlers JS asociados

Qué es una función handler JS

Antes de comenzar con los ejemplos, debemos introducir una serie de definiciones sobre el contexto de esta problemática.

Previamente en este capítulo se introdujo el concepto de "*event handler*" o **manejador de eventos**, que son las funciones que responden a los eventos que se producen. Los **eventos** son la manera que se tiene en JavaScript de controlar las acciones del usuario y poder definir un comportamiento de la página cuando estos se produzcan.

Por ejemplo, cada vez que el usuario realiza una acción interceptable, como hacer clic o pulsar una tecla, decimos que se dispara un evento generado por el usuario. Además de eventos disparados por el usuario, existen eventos disparados por el sistema operativo o por el navegador. Por ejemplo, cuando termina la carga de una página web en el navegador se dispara el evento de sistema *onload* (carga terminada).

Se pueden definir handlers de eventos de tres formas diferentes:

1. Código JavaScript dentro de un atributo del propio elemento HTML.
2. Definición del evento en el propio elemento HTML pero el manejador es una función JS externa.
3. Manejadores semánticos asignados mediante DOM sin necesidad de modificar el código HTML de la página.

Cualquiera de estos tres modelos funciona correctamente en todos los navegadores disponibles en la actualidad. Las formas 1 y 2 son las que nos interesan a nosotros, ya que son las más comunes en los ejemplos que veremos a continuación.

4.3.1 Ejemplo #1 Sitio para compra de autos Karvi

Sitio: https://www.karvi.com.ar/autos-nuevos-0km/

Sección para compra de autos nuevos

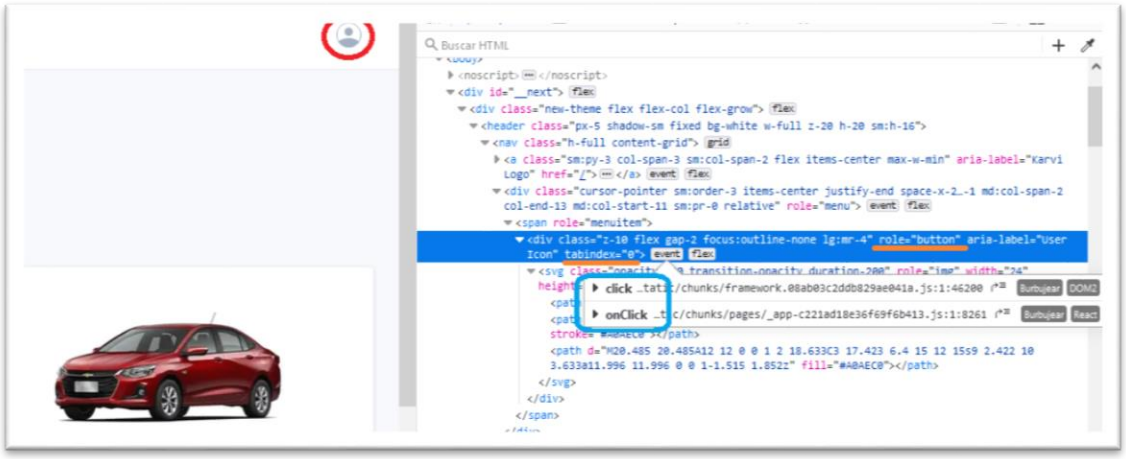


Figura 4.3.1: Captura del código del “botón” para iniciar sesión en la página.

Comportamiento con NVDA

Como observamos en la Figura 4.3.1, se encuentra resaltado el elemento para poder iniciar sesión en la página el cual es inaccesible. Este elemento tan importante para que el usuario pueda realizar gran cantidad de acciones en la página, termina siendo ignorado por NVDA.

Como se puede ver en su código, el elemento es un * * el cual posee tanto el rol de botón, tabindex en 0 y dos handlers JS asignados. Aun así, con todos estos atributos para disfrazar al div como un botón no fueron suficientes para que deje de comportarse como un * ,* ya que NVDA lo ignoró por ser un div y no un botón.

Otro motivo por el cual el elemento es ignorado por el screen reader, es por su contenido. Este div posee un elemento SVG que es un elemento que también suele ser ignorado.

4.3.2 Ejemplo #2 Sitio de contenido multimedia GENIUS

Sitio: https://genius.com/

Página Home principal de noticias y contenido multimedia

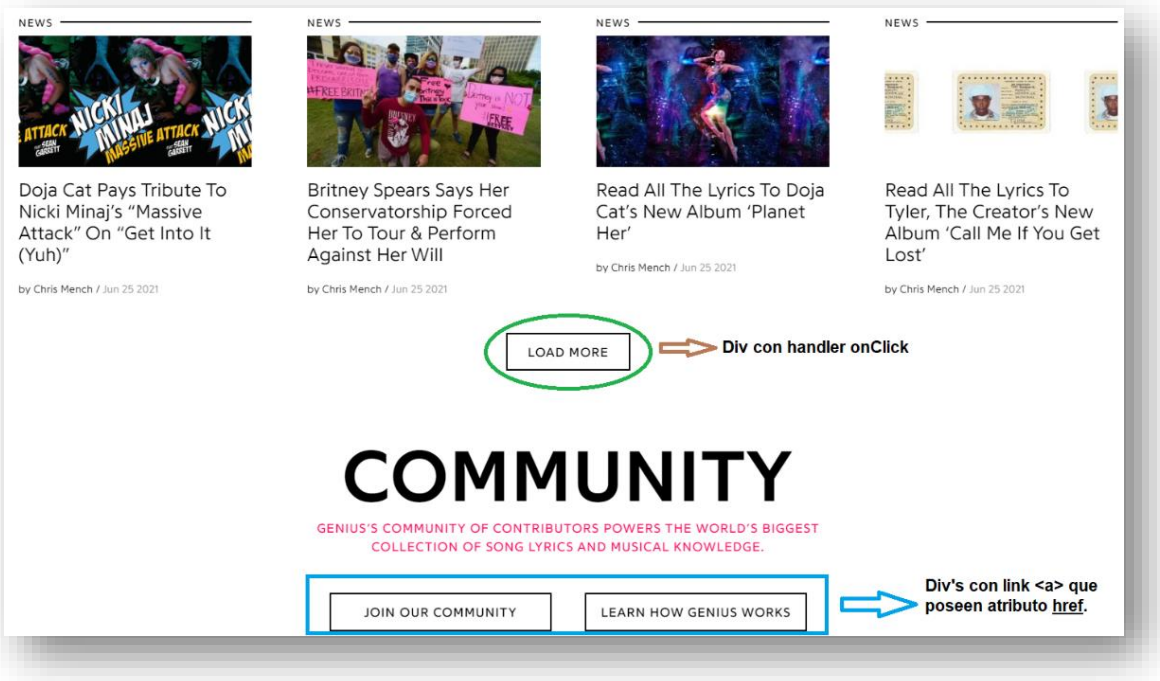


Figura 4.3.2: Captura que contiene tres elementos definidos de maneras diferentes.

Comportamiento con NVDA

Mientras se navega con NVDA a través de las noticias de la página, se llega a la sección que se muestra en la Figura 4.3.2, donde se observa tres botones. Aunque no lo son, son div's simulando ser botones. Aun así, este no es el principal problema, sino que el botón "Load More" es inaccesible. Esto es debido a que es un elemento que posee un *handler onclick* asignado para mostrar más noticias, con tener solo asignado este handler no va a hacer que el screen reader lo detecte.

Caso contrario sucede con los div's "botones" *Join Our Community* y *Learn How Genius Works*, ya que en este caso NVDA detecta que estos elementos contienen links <a> con sus respectivos atributos HREF. Como sabemos este atributo es especifico de los links y contiene el destino de redirección de ese link, si un link no tiene este atributo va a ser ignorado por los screen readers.

4.3.3 Ejemplo #3 Links inaccesibles en el sitio web de Garbarino

Sitio:

- https://www.garbarino.com/productos/tecnologia
- https://www.garbarino.com/producto

La página de Garbarino tiene varios ejemplos de elementos disfrazados de links que son inaccesibles. A continuación, se mostrarán dos secciones diferentes de Garbarino.com detallando los casos de elementos que generan problemas de accesibilidad.

Página de la categoría Tecnología de Garbarino.com

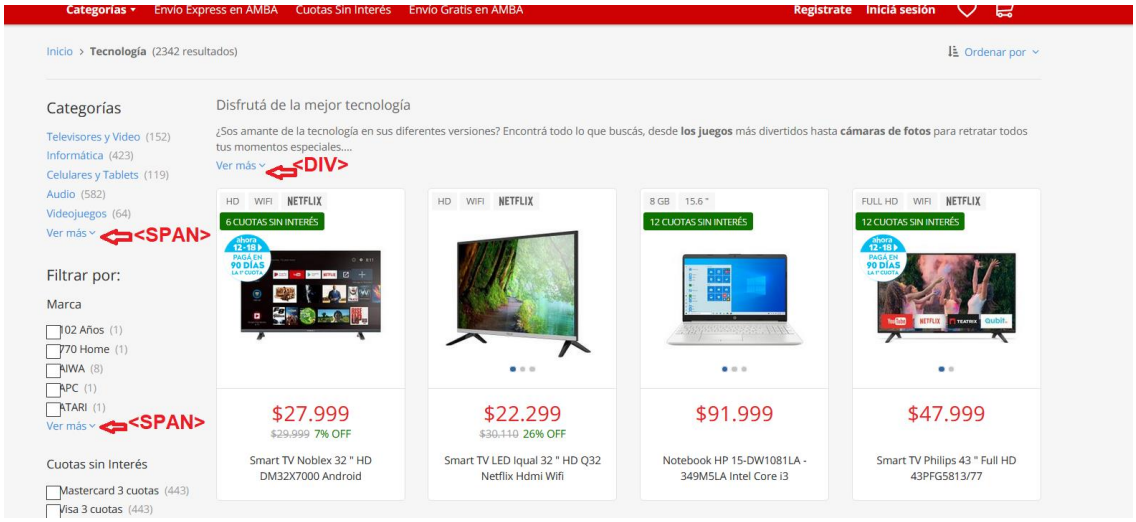


Figura 4.3.3: Captura de Garbarino.com que muestra resaltado tres ejemplos de elementos inaccesibles en el sitio.

Comportamiento con NVDA

En la Figura 4.3.3 se encuentran señalados los elementos “Ver más”, que visualmente parecen links, pero no lo son. Están definidos con elementos ** y * * y además para realizar la acción de mostrar más ítems tiene asignados handlers JS. Por lo tanto, debido a que no fueron declarados como <a> (link) son ignorados por NVDA.

Es bueno aclarar que los elementos de texto son ignorados por los screen readers durante la navegación mediante la tecla *TAB*.

Página de producto para comprar de Garbarino.com

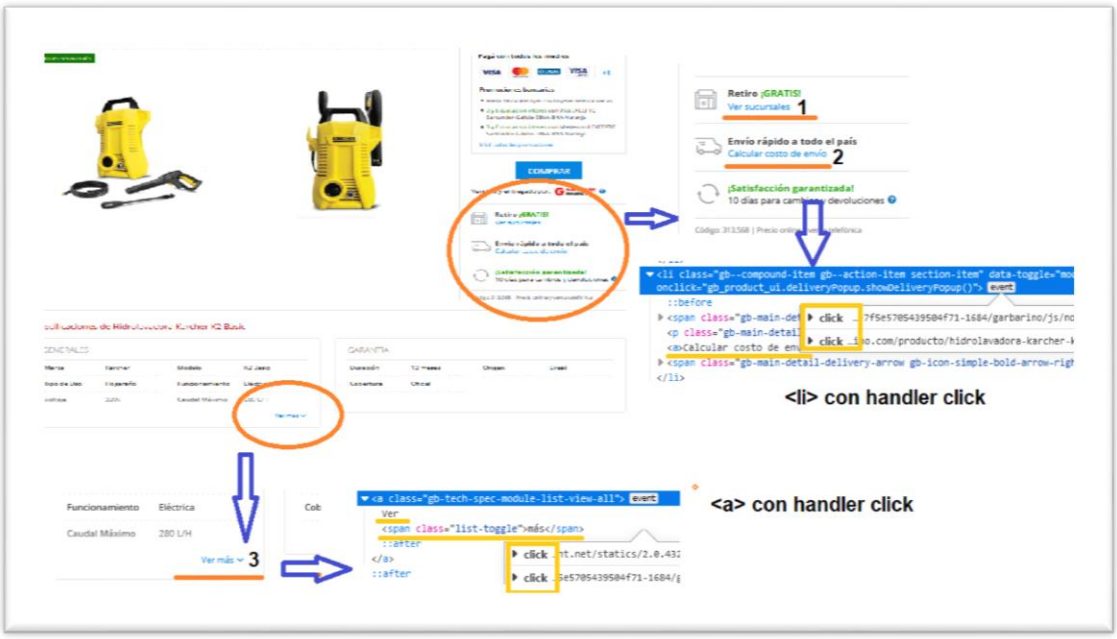


Figura 4.3.4: Captura de Garbarino.com que muestra tres ejemplos de elementos inaccesibles en el sitio de comprar producto.

Comportamiento con NVDA

En ese caso también tenemos elementos que simulan ser links y no son capturados por NVDA durante la navegación. Como se puede observar en la Figura 4.3.4, los elementos 1 y 2 son links <a> sin atributo href, de la redirección se encarga su elemento padre de tipo listado <li> el cual tiene asignado un handler click. Ósea que en este ejemplo se intentó simular el comportamiento de un link en su elemento padre, pero fue ignorado por NVDA.

En el elemento 3, sucede casi lo mismo, pero en este caso se intentó simular un link utilizando un elemento con un handler click para mostrar más elementos.

En ambos grupos de elementos, se podría haber logrado tener elementos que realicen la misma acción mediante el uso apropiado de links y poder ser accesibles.

4.4 Herramienta para la detección y reporte de elementos interactivos inaccesibles

Como se ha venido mencionando a lo largo de este capítulo, se han detectado varios casos de elementos interactivos que han sido ignorados por el screen reader. Por ende, los podemos catalogar como elementos inaccesibles y que generan problemas de accesibilidad o *accessibility smell*s en la página.

Cuando hablamos de **elementos interactivos** nos referimos a aquellos con los cuales el usuario puede interactuar y agregan algún tipo de funcionalidad a la página.

4.4.1 Introducción a la detección de elementos interactivos inaccesibles

Como se ha mencionado anteriormente, durante la búsqueda manual de *accessibility smells* se han encontrado varios casos de elementos que simulan ser otros mediante el uso de *handlers JavaScript*. Por ejemplo:

- Botones desarrollados con DIV's.
- Links <a> sin el atributo HREF con forma de botón.
- Elementos que simulan ser links.

Visualmente parecen links y botones, los cuales para usuarios sin problemas de visión se ven y funcionan como tal. Sin embargo, al utilizar algún programa de asistencia como un lector de pantalla, estos elementos son ignorados generando que el usuario:

- en muchos casos no pueda continuar con el flujo normal de la página. Por ejemplo, un botón de *submit* de formulario que no puede ser utilizado por NVDA.
- Mientras que, en otros casos, el usuario se pierda de contenido útil de la página.

No obstante, no todos estos elementos terminan siendo inaccesibles, ya que algunos desarrolladores siempre tienen en mente la accesibilidad durante el proceso de desarrollo. Y mediante la utilización de atributos ARIA y el uso consciente de JavaScript en el código, logran desarrollar **elementos interactivos accesibles**.

4.4.2 Descripción del método de detección

El primer paso para la detección de este tipo de elementos inaccesible, consiste en encontrar aquellos elementos que tengan *handlers JS* asignados. Por ejemplo:

```
<div class="boton" onclick="enviarFormulario()">Enviar</div>
```

Para este caso podemos utilizar la DOM API, ya que con ella podremos acceder a atributos del elemento y podremos ver que tiene el atributo onclick presente. Sin embargo, lamentablemente durante nuestras pruebas se detectaron muy pocos casos de elementos que tienen sus handlers definidos de esta manera.

Principal problema: handlers definidos externamente

La gran mayoría de elementos interactivos inaccesibles tienen sus handlers JS definidos en archivos .js externos, los cuales son importados al código HTML de la página web para poder aplicar la lógica definida en estos archivos. Generalmente efectúan cambios en el DOM de la página agregando, eliminando y modificando elementos dentro de la misma. El mismo código JS es interpretado y ejecutado por el propio navegador web.

Como se logra observar en la Figura 4.4.1, la estructura básica de una página web puede estar compuesta por tres tipos de archivos diferentes:

- **Archivos HTML:** se define la estructura de la página, donde están definidos todos sus componentes HTML.
- **Archivos CSS:** contienen los estilos que se aplican a los componentes de la página.
- **Archivos JS:** contienen las funcionalidades definidas con JavaScript que se ejecutan en la página,

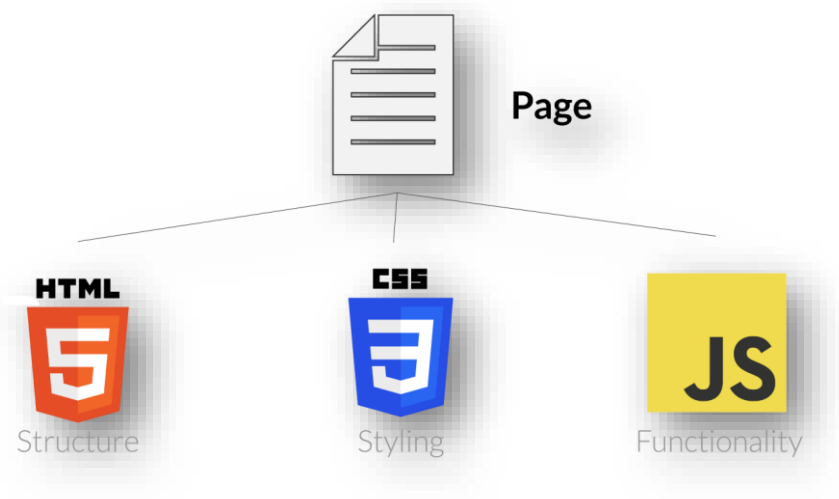


Figura 4.4.1: Figura utilizada para explicar la relación entre archivos HTML, CSS y JS.

Cabe destacar que tanto el contenido de los archivos .css y .js pueden ser definidos dentro del archivo *.html*. Por ende, no es obligatorio definir estos archivos. Aun así, generalmente los desarrolladores escriben el código HTML, CSS y JS por separado.

Parser de código JavaScript

Debido a que el código JS generalmente está definido en **scripts JS externos**, fue necesario encontrar la forma de revisar estos archivos JS en busca de handlers asignados a elementos de la página web. En el informe de avance de esta tesina, se mencionó que estaba en proceso de desarrollo una herramienta para poder parsear los archivos JS asociados a un sitio web y así lograr aplicar una búsqueda de estos handlers. Para ello se realizó una investigación de posibles herramientas de parseo de código JS, en la cual no se logró hallar ninguna herramienta disponible para esta tarea.

De este modo, se continuó con la implementación de un algoritmo que se encargue de esto. Este algoritmo permitió acceder a los archivos JS y realizar búsquedas de handlers asociados a elementos de la página. La detección estaba basada en analizar los scripts convertidos a strings y realizar una búsqueda de patrones similares a las formas de declarar handlers. Por ejemplo, se buscaban patrones como: ***addEventListener('click')*** ó ***.click(...)***, etc.

En la Figura 4.4.2 se muestra el código de estos ejemplos para un mejor entendimiento, pero observando estos ejemplos, ya se puede entender que son varias las formas de declarar un simple *handler onClick* y esto va a significar un gran problema a la implementación de nuestro algoritmo de búsqueda.

```
<div id="myBtn" class="boton" onclick="enviarFormulario()">Enviar</div>

// código JavaScript definido en scripts.js
let element = document.getElementById("myBtn");
element.addEventListener("click", myFunction);

// handler click en AngularJS
angular.element(document.querySelector('.boton')).bind('click', function () {
  myFunction();
});
// handler click usando JQuery
// JQuery forma #1
$(".boton").on('click', function () {
  myFunction();
});
// JQuery forma #2
$("#myBtn").click(function () {
  myFunction();
});

function myFunction() {
  alert("Hello World!");
}
```

Figura 4.4.2: Código de ejemplo para mostrar distintas implementaciones de handlers de tipo click.

Se probó la detección satisfactoria en páginas simples, pero cuando se comenzó a testear la búsqueda en sitios más complejos, nos dimos cuenta que existen muchas más formas de declarar un handler de las que pensábamos. Esto se debe a la gran popularidad que ganó JavaScript estos últimos años en lo que es el desarrollo web. Hay infinidad de nuevos frameworks basados en JavaScript, donde cada uno de ellos tiene una forma diferente de declarar estos handlers.

A medida que se iban encontrando nuevas formas de declarar handlers JS, se iban agregando al código del algoritmo de búsqueda. Sin embargo, llego un punto que esto era algo insostenible y muy ineficiente a futuro. Así que se decidió no continuar con este método.

La solución: getEventListeners

Buscando otras opciones para la detección de handlers JS, se descubrió que es una problemática que afecta también a desarrolladores web y no han encontrado la forma de resolverlo. Hay escenarios donde necesitan conocer si un elemento determinado tiene asociada una función JS.

Es así como se llegó a la página de documentación para desarrolladores de Google Chrome [ChromeDevTools], la cual se va actualizando a medida que se van desarrollando nuevas funcionalidades en JavaScript soportadas por el web browser.

En esta documentación se descubrió la función llamada: **getEventListeners(object)**, la cual retorna los event listeners registrados en un objeto especifico. El objeto que devuelve esta función contiene un arreglo con cada uno de los tipos de eventos (click o focus, por ejemplo). Cada elemento dentro de este arreglo contiene todos los listeners definidos para ese evento determinado, por ejemplo: si el elemento *document* posee tres listeners para eventos de click, en el arreglo se van a listar los tres listeners bajo el elemento “click” como podemos ver en la Figura 4.4.3.

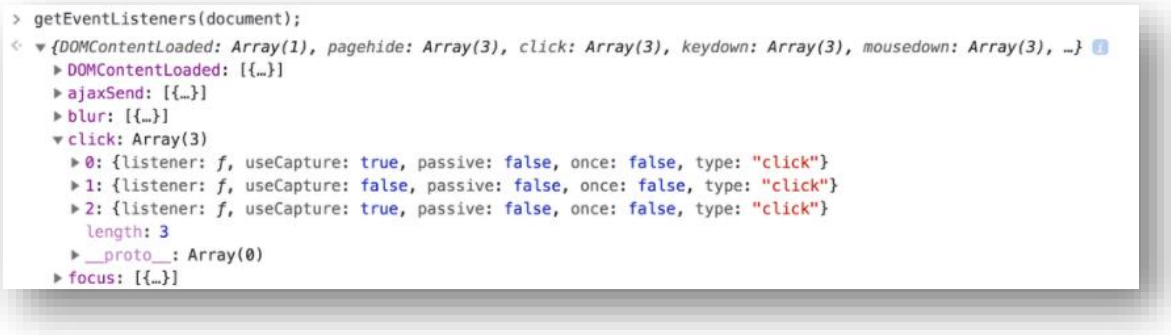


Figura 4.4.3: Captura de la consola de Chrome donde se ejecutó la función *getEventListeners* para el elemento *document*.

Por lo tanto, luego de este descubrimiento, ya no iba a ser necesario trabajar en ningún tipo de parser para JavaScript. El mismo browser nos provee una función para obtener todos los listeners asociados a cualquier elemento HTML dentro de la estructura del DOM de cualquier página web.

4.4.3 Implementación del finder para la búsqueda de elementos interactivos inaccesibles

Una vez resuelto y definido como va a desarrollarse la implementación del método para la detección de los handlers JS de un elemento HTML, el paso siguiente consistió en comenzar a desarrollar el algoritmo del **finder** para realizar esta búsqueda.

La función *getEventListeners(object)* se aplica a un elemento específico, por lo tanto, se debe definir a cuál tipo de elemento se necesita conocer si tiene o no handlers JS asignados. La razón es que no tendría sentido y sería muy costoso recorrer todos los elementos de la página, ya que una página web puede tener hasta más de 15 mil elementos.

Durante las pruebas manuales buscando problemas de accesibilidad en distintas páginas web, se fue recolectando información de cuáles son los tipos de elementos HTML que más generaban este tipo de problema de accesibilidad, y encontramos este problema en elementos HTML de tipo: **div**, **span**, **li** y **a**. Por lo tanto, se centró la búsqueda en solo este tipo de elementos.

Entonces se desarrolló una primera versión del *finder*. Durante las primeras pruebas del algoritmo descubrimos el primer problema de este método, el cual fue que el método *getEventListeners* solo funcionó correctamente en el browser de Mozilla Firefox. Esto es debido a que en Google Chrome solo se puede ejecutar esta función con la Dev Tool de la consola del browser, en cambio en Firefox, este método funciono correctamente tanto en el código de la extensión web como en la consola del mismo navegador. Se espera que pronto pueda estar habilitada en Google Chrome para ser utilizada también en scripts.

La sección principal del código del *finder* realiza dos acciones principales:

1. Búsqueda de handlers declarados en el código del elemento HTML. Ejemplo:

   * *

2. Búsqueda de handlers utilizando la función *getEventListeners*. Estos handlers pueden estar declarados como código JS dentro de tags <script> en el mismo documento .html de la página o en archivos .js externos.

Una vez que se detectó si un elemento de la página posee un handlers asignado, lo almacenamos junto a diversa información del elemento.

No obstante, esto no es el final de la definición de nuestro finder:

> *Que un elemento web tenga handlers JS asignados no significa que sea inaccesible, pero puede ser un indicio de que lo es. Hay que revisar ciertas características del mismo para afirmar que puede generar algún smell de accesibilidad.*

Por esto, los elementos que poseen handlers JS son almacenados en un arreglo denominado *elementos_con_handler*, el cual está asociado a la URL de la página. Es decir, que cada sitio web por donde se ejecuta la extensión web posee su propio arreglo con estos elementos. Además, no solo se guarda información como el tipo de elemento (DIV, SPAN, A, …), su ID, clases, etc., sino también los valores de varios atributos utilizados para brindar accesibilidad como lo son: *tabindex*, *aria_label* y *role*(ARIA).

Sin embargo, también se necesitó almacenar información sobre los hijos del elemento, ya que hay casos donde el elemento catalogado como inaccesible, termina siendo accesible para los screen readers ya que posee hijos que son accesibles.

Por ejemplo: se detectó un elemento DIV con un handler *onclick* asignado, no posee ningún tipo de atributo que lo haga accesible (aria, rol, etc..), por lo tanto, debería ser ignorado por el screen reader ya que no brinda ningún tipo de contenido. Pero este elemento posee un elemento hijo o "child" de tipo link <a> con su respectivo atributo href. Como se sabe, todo elemento <a> que posea un destino definido en su atributo href va a ser leído correctamente por el screen reader. Lo mismo sucede con los elementos de tipo <button>, no necesitan ningún tipo de atributo de accesibilidad para que sean leídos por los screen readers como NVDA.

Anteriormente en la Figura 4.3.2, observamos un claro ejemplo de eso. Dos botones que se ven iguales, pero uno es accesible y el otro no, y esto se debe a que uno es solo un elemento *DIV* con *onclick*, y el que es accesible es un *DIV* que contiene un *<a>* con *href*.

Volviendo a la implementación del *finder*, para lograr abarcar los diferentes casos de implementación de los elementos con *handlers JS* y no reportar falsos positivos, la estructura de la información que se va a almacenar contiene tanto los atributos del elemento como de sus elementos hijo. Como se puede observar en el diagrama de la Figura 4.4.4, donde cada elemento (**ElementWithHandler**) posee diferentes arreglos con esta información.

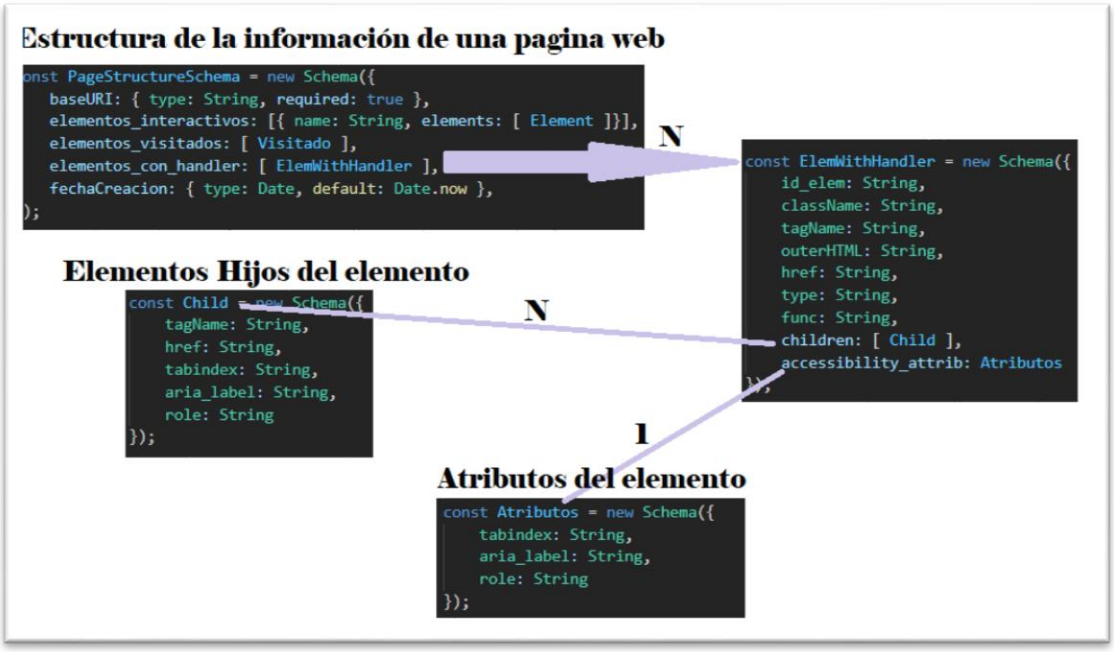


Figura 4.4.4: Diagrama de la estructura de los datos que se manejan para la detección de elementos inaccesibles con handlers JS.

Toda esta información reportada por la extensión web, va a quedar almacenada en la base de datos. Luego, esta va a ser consumida por la aplicación de reportes.

Para este finder, se decidió que la lógica de analizar y reportar si un elemento es accesible o no, lo va a realizar la misma aplicación de reportes. Esta lógica también se podría haber realizado tanto en la misma *web extension* como en la *API REST*.

Una vez que la aplicación de reportes recibe una petición para generar un reporte de problemas de accesibilidad en una página determinada, esta analiza la información almacenada, y luego genera los reportes de accesibilidad. Esto lo veremos en los ejemplos a continuación, donde se realizó la ejecución de la herramienta sobre varios sitios web y se mostraran los reportes resultantes.

Validación extra

A priori, el finder desarrollado en este capítulo se basa en la búsqueda de aquellos elementos que simulan ser otros con la ayuda de handlers JS. Mediante la búsqueda de elementos con handlers asignados a ellos y el análisis de sus atributos, se ha logrado detectar una gran cantidad de elementos de este tipo en varios sitios web. En la sección 4.5 se citarán varios de estos ejemplos detectados por la herramienta de detección.

Además de esto, para mejorar la precisión del finder, se incorporó un método para reportar los elementos que recibieron alguna interacción por parte del usuario. Para este caso, nos interesó detectar los elementos que han recibido el foco desde teclado, esto no significa que son elementos accesibles, pero nos da la pauta de que el screen reader detectó a ese elemento y se lo comunicó al usuario.

Este método también fue implementado en la extensión web y su función principal es la de escuchar los eventos **"onfocus"** que suceden en determinados elementos de una página web. Para este trabajo, se focalizó la detección en elementos de tipo link <a> a modo de analizar la efectividad junto al finder de elementos con handlers JS.

Los eventos de *foco* son "escuchados" a través de elementos de tipo **listeners JavaScript,** que son asignados a cada uno de los elementos de tipo link. Por lo tanto, cada vez que un usuario posicione el foco de teclado en el elemento, se va a disparar una función JS. Esta función lo único que hace es enviar a la API REST la información del elemento que recibió el foco.

Además, en la base de datos de la herramienta se almacenará una estructura básica de todos los links de cada sitio web por donde se ejecutó la herramienta. Esto servirá para mencionar en el reporte cuales elementos han recibió el foco y cuáles no. Con esto se busca darle al usuario que va a leer el reporte, un indicio de que cuales elementos pueden tener algún problema de accesibilidad.

Con este método de análisis extra se busca no solo ofrecer un reporte de aquellos elementos que pueden ser inaccesibles (porque fueron declarados sin utilizar el debido tipo de elemento HTML y sus correspondientes atributos para lograr accesibilidad), sino también ofrecer una validación extra de si pueden recibir el foco mediante el uso de teclado. Esta solución surgió en base uno de los métodos utilizados en trabajo realizado por Antonelli [Antonelli2018], el cual fue descrito en el capítulo 2. En el mismo se utilizaron listeners para chequear si el usuario puede utilizar los elementos internos de un menú desplegable.

4.5 Resultado final de la herramienta para la detección de elementos inaccesibles con handlers JS

Antes de continuar con los resultados, en la Figura 4.5.1, se describirán los distintos componentes de la estructura de los reportes generados por la herramienta.

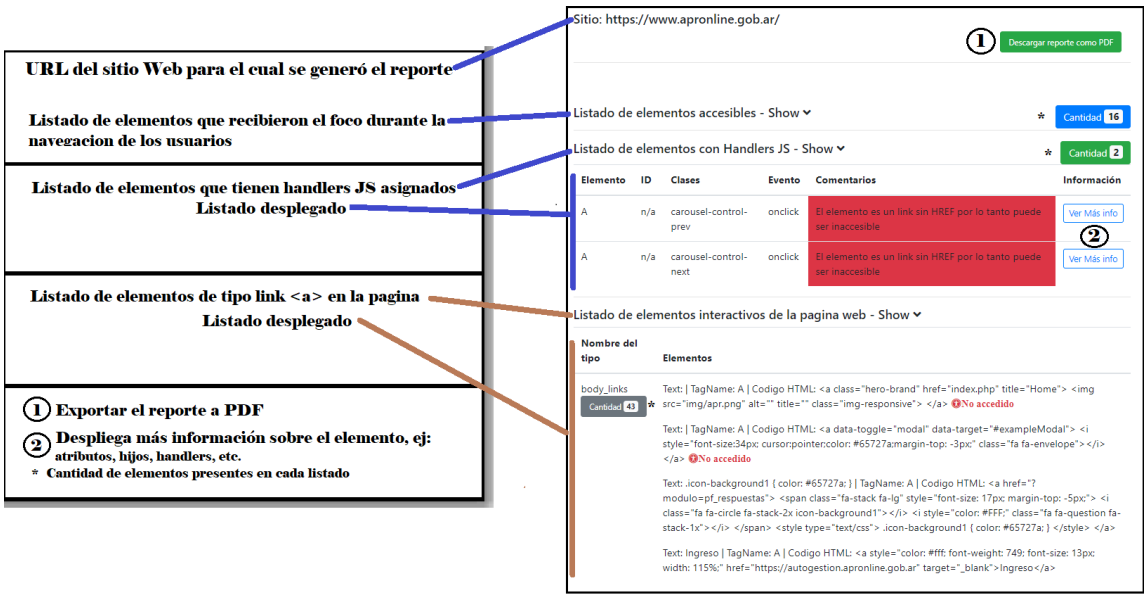


Figura 4.5.1: Diagrama explicativo de cómo están compuestos los reportes generados por la herramienta para la detección elementos interactivos inaccesibles

Como se puede observar en la Figura 4.5.1, en el listado de elementos con handlers JS al lado de cada elemento hay un botón de “Ver más info”. El cual despliega más información útil sobre cada uno de los elementos, como lo es su id, clases, la función handler JS que tiene asociada, tipo de elemento, listado de sus elementos hijos (si los tiene), etc. En la Figura 4.5.2 se muestra un ejemplo:

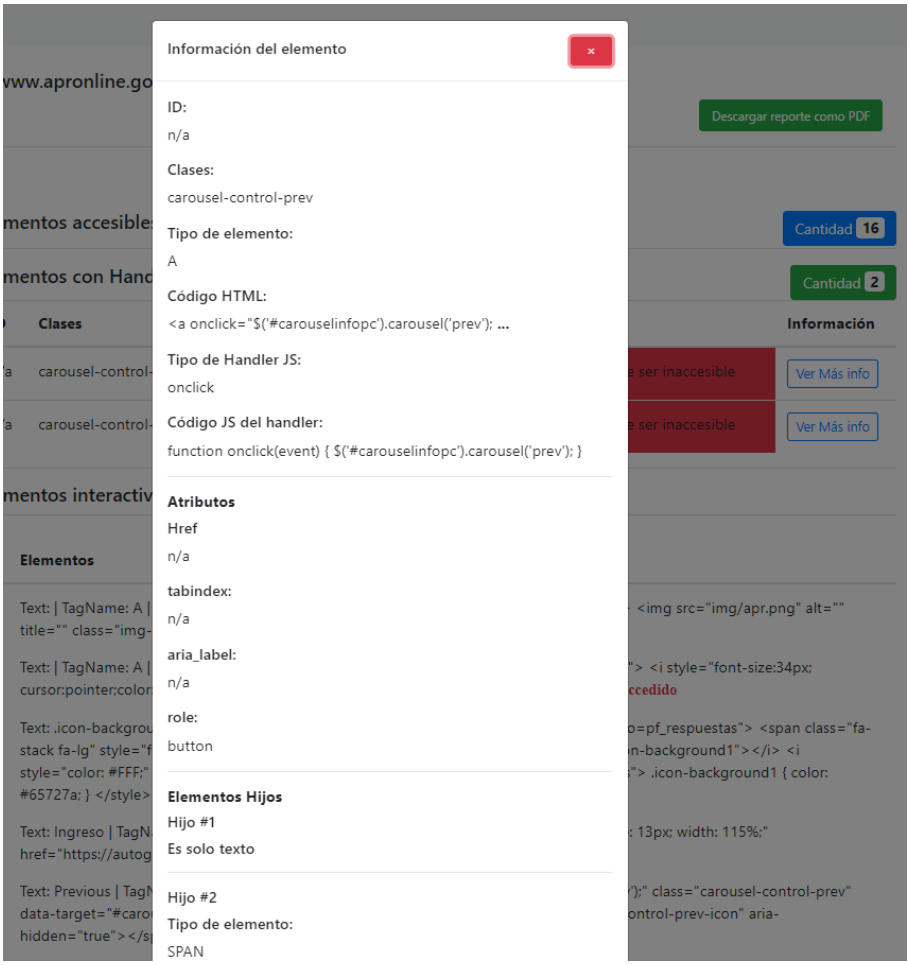


Figura 4.5.2: Captura del modal con más información sobre un elemento reportado.

Además, en el mismo listado de elementos se decidió listar a todos los elementos que poseen handlers JS asignados. Por lo tanto, para diferenciar, en la columna de “Comentarios” están remarcados con color verde aquellos elementos que no poseen ningún problema de accesibilidad y con color rojo aquellos que pueden presentar problemas.

Para mostrar el resultado final de la herramienta para la detección de elementos web inaccesibles con handlers JS, se ejecutó la misma en varios sitios web, con los siguientes resultados.

4.5.1 Sitio web con elementos que poseen handlers JS #1

Página de principal del portal de APR

URL

www.apronline.gob.ar

Capturas

Figura 4.5.3: Screenshot de los elementos para controlar el scroll del carousel.

Comportamiento de NVDA

En este caso el screen reader *NVDA* no mencionó ninguno de los dos elementos que están señalados en la Figura 4.5.3. Por lo tanto, el usuario no puede utilizar el carousel de trámites.

Reporte final de la herramienta

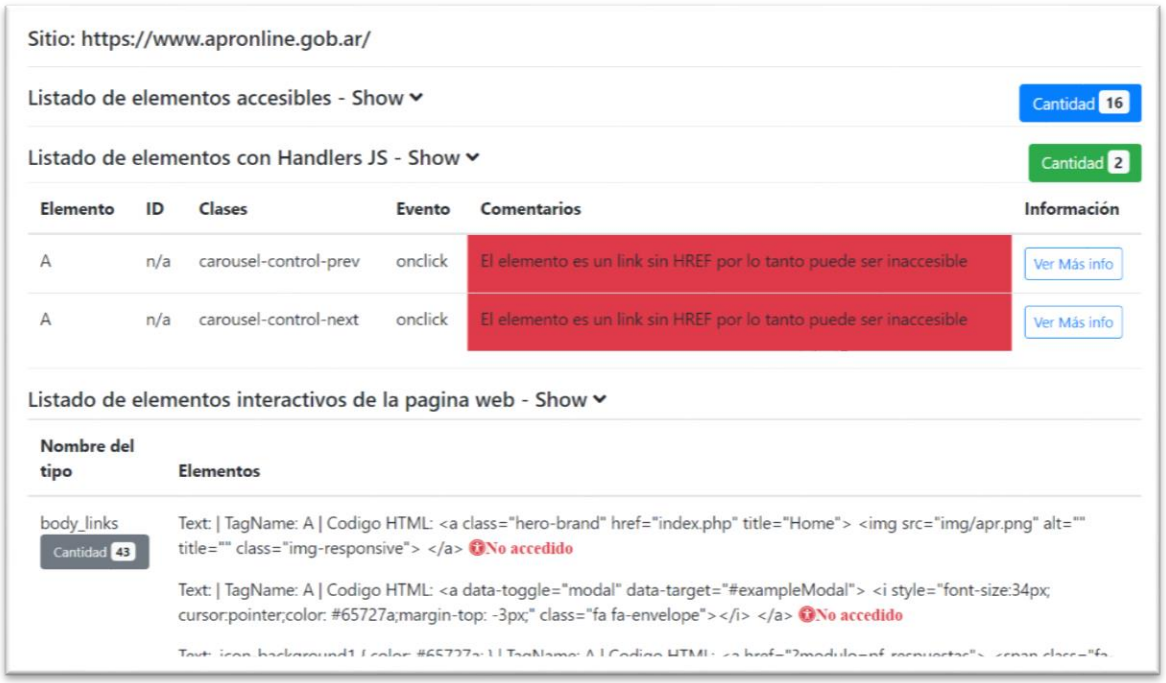


Figura 4.5.4: Reporte generado por la aplicación de reportes para la página principal de APR.

Como se puede observar en la Figura 4.5.4, se reportaron ambos elementos del carousel. Los cuales son links <a> con handlers JS asignados y sin atributo HREF ni atributos para lograr accesibilidad.

4.5.2 Sitio web con elementos que poseen handlers JS #2

Página de compras de Garbarino.com

URL

www.garbarino.com/producto/hidrolavadora

www.garbarino.com/productos/televisores-y-video

Capturas

El reporte para ese caso es bastante amplio, donde se reportaron varios elementos inaccesibles. Por lo tanto, dividiremos el análisis por grupo de elementos según aparezcan en el sitio web.

Notificaciones del sitio

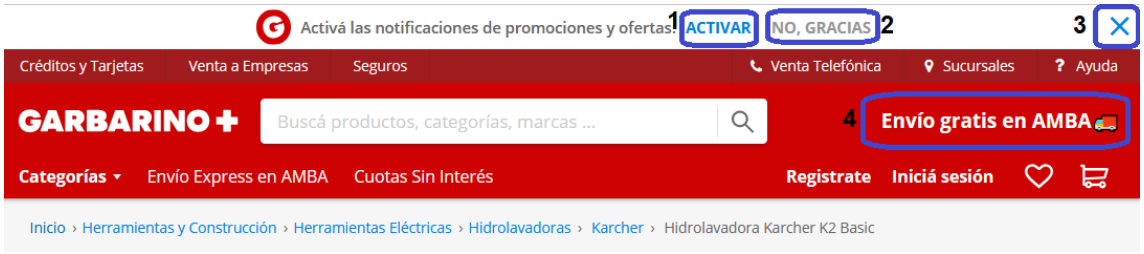


Figura 4.5.5: Screenshot que contiene el primer conjunto de elementos con handlers JS.

Links para ver sucursales y cálculo del envío

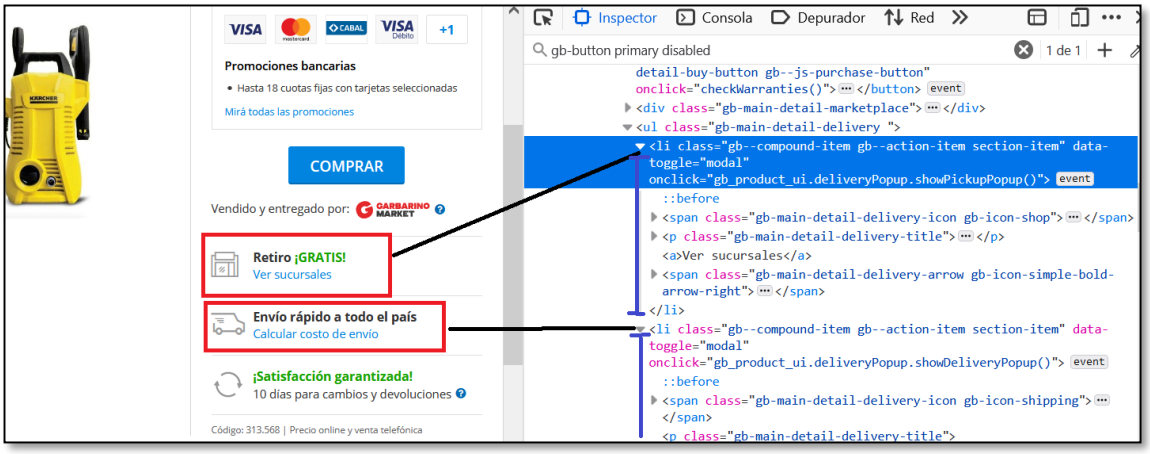


Figura 4.5.6: Screenshot que muestra un grupo de dos elementos de tipo <li> con handlers JS.

Menú de categorías de tipo drow-down

Figura 4.5.7: Diagrama que muestra un menú desplegable hecho con elementos <li> y handlers JS, junto a su respectivo código HTML.

Pequeño menú drop-drown y links de ver más items

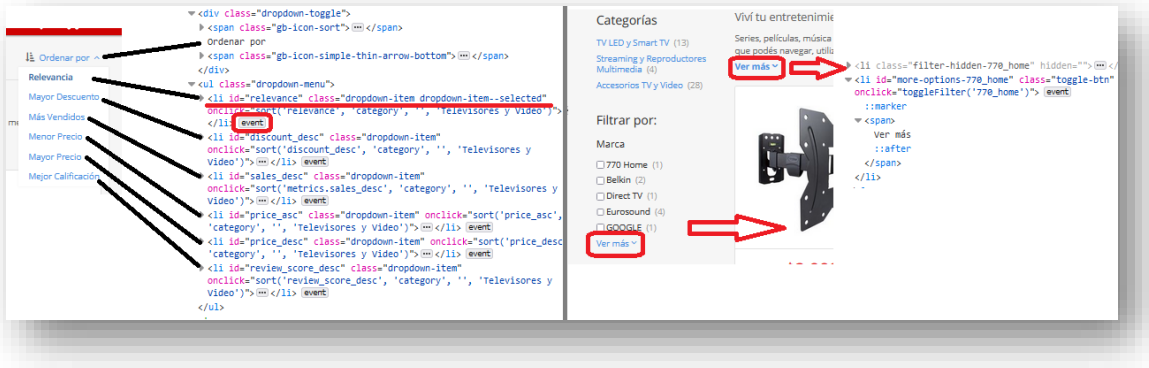


Figura 4.5.8: Diagrama que muestra cómo están declarados en el código un menú desplegable de links y otros dos links para visualizar más elementos de un listado.

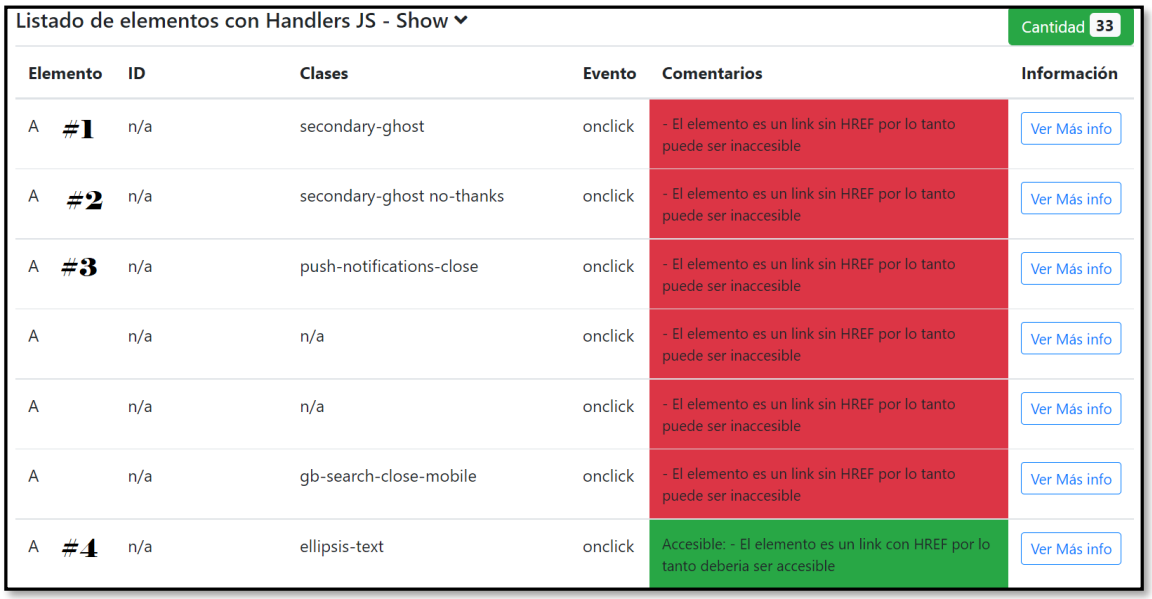
Comportamiento de NVDA

En la Figura 4.5.5, tanto los elementos 1, 2 y tres no fueron nombrados por NVDA ya que no es posible posicionarse en ellos mediante el uso de teclado. Se intentó hacerle foco al elemento mediante la tecla TAB y también usando el *modo revisión* de NVDA para poder utilizar los atajos rápido de teclado: K y K+SHIFT para moverse entre los enlaces de la página, pero ningún método funciono. El elemento #4 también posee handlers JS, pero si logro capturar el foco y así ser nombrado por NVDA, cuando analicemos el reporte de la herramienta diremos el porqué.

Los elementos señalados en las siguientes figuras (4.5.6, .7 y .8) tampoco fueron mencionadas por NVDA.

Reporte final de la herramienta

Notificaciones del sitio



Listado de elementos con Handlers JS - Show

Cantidad 33

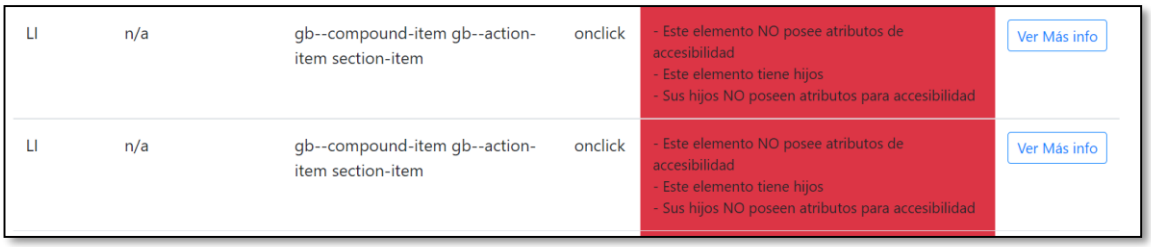
Elemento	ID	Clases	Evento	Comentarios	Información
A #1	n/a	secondary-ghost	onclick	- El elemento es un link sin HREF por lo tanto puede ser inaccesible	Ver Más info
A #2	n/a	secondary-ghost no-thanks	onclick	- El elemento es un link sin HREF por lo tanto puede ser inaccesible	Ver Más info
A #3	n/a	push-notifications-close	onclick	- El elemento es un link sin HREF por lo tanto puede ser inaccesible	Ver Más info
A	n/a	n/a	onclick	- El elemento es un link sin HREF por lo tanto puede ser inaccesible	Ver Más info
A	n/a	n/a	onclick	- El elemento es un link sin HREF por lo tanto puede ser inaccesible	Ver Más info
A	n/a	gb-search-close-mobile	onclick	- El elemento es un link sin HREF por lo tanto puede ser inaccesible	Ver Más info
A #4	n/a	ellipsis-text	onclick	Accesible: - El elemento es un link con HREF por lo tanto deberia ser accesible	Ver Más info

Figura 4.5.9: Captura parcial del reporte generado por la herramienta donde se encuentran reportados los elementos señalados en la Figura 4.5.5.

Como se puede observar en la Figura 4.5.9, se encuentran reportados los elementos 1, 2 y 3 de la Figura 4.5.5 como elementos inaccesibles. Mientras que el #4 está marcado como accesible, y esto es correcto ya que del os 4 elementos, solo el #4 posee el atributo *href*. Y como se ha mencionado anteriormente, NVDA ignora a aquellos links que no poseen este atributo, ya que es un atributo necesario para el correcto funcionamiento de los elementos links <a>.

Links para ver sucursales y cálculo del envío

Ambos elementos fueron reportados ya que poseen handlers onclick como se logra observar en la Figura 4.5.10. Además, fueron catalogados como inaccesibles correctamente, esto es correcto ya que fueron ignorados por NVDA y no poseen ningún tipo de atributo para lograr accesibilidad.



LI	n/a	gb--compound-item gb--action-item section-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	n/a	gb--compound-item gb--action-item section-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info

Figura 4.5.10: Captura parcial del reporte generado por la herramienta donde se encuentran reportados los elementos señalados en la Figura 4.5.6.

Menú de categorías de tipo drow-down

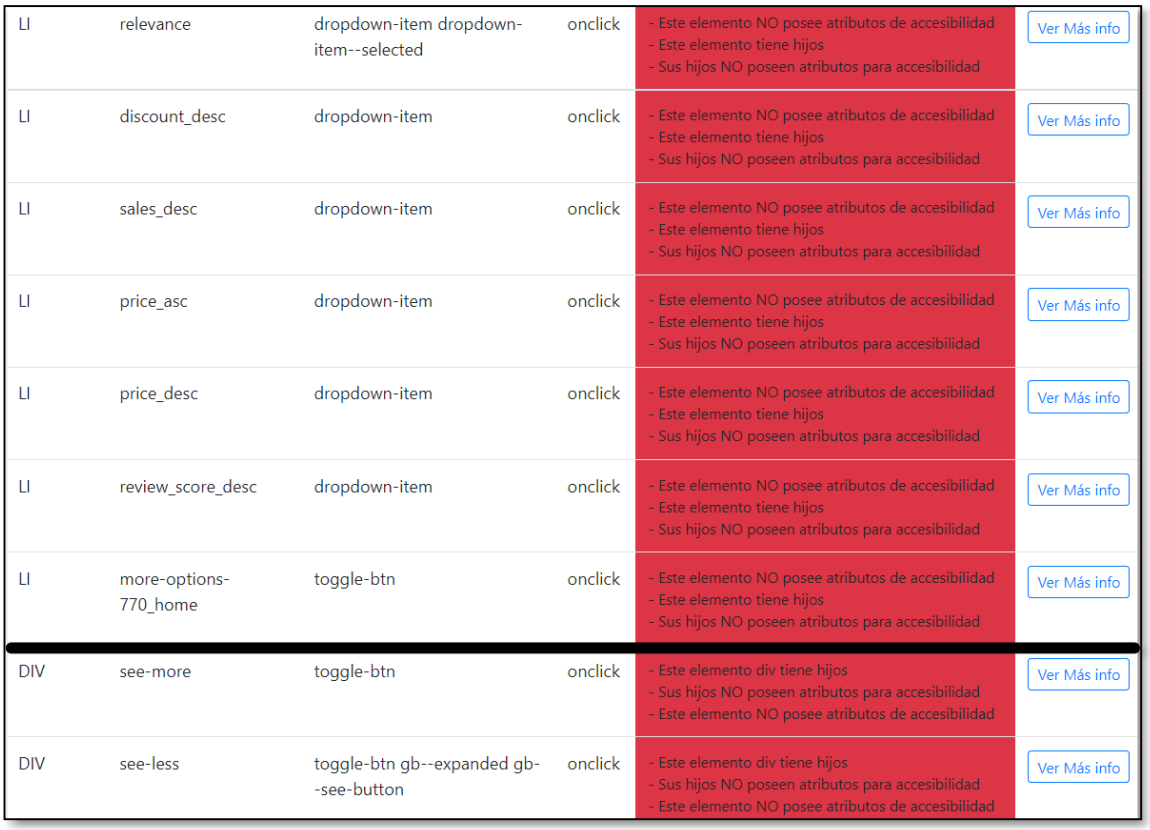
Como se puede observar en la Figura 4.5.10, el reporte incluye a los nueve elementos de tipo <li> que componen al menú desplegable de categorías. También fueron catalogados como inaccesibles dando detalles de los motivos en el mismo reporte. En este caso también funcionó correctamente la detección.

Figura 4.5.10: Captura parcial del reporte generado por la herramienta donde se encuentran reportados los elementos señalados en la Figura 4.5.7.

Pequeño menú drop-drown y links de ver más ítems

Los elementos mencionados en la Figura 4.5.8, se encuentran en diferentes secciones de la página de Garbarino. Por lo tanto, también se incluyeron en esta sección de ejemplos.

Como se puede observar en la Figura 4.5.11, fueron reportados como inaccesibles tanto a los elementos <li> que componen al menú drop-down como a los 's que simulan ser links. En este caso también la detección fue correcta.



LI	relevance	dropdown-item dropdown-item--selected	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	discount_desc	dropdown-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	sales_desc	dropdown-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	price_asc	dropdown-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	price_desc	dropdown-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	review_score_desc	dropdown-item	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
LI	more-options-770_home	toggle-btn	onclick	- Este elemento NO posee atributos de accesibilidad - Este elemento tiene hijos - Sus hijos NO poseen atributos para accesibilidad	Ver Más info
DIV	see-more	toggle-btn	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info
DIV	see-less	toggle-btn gb--expanded gb--see-button	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info

Figura 4.5.11: Captura parcial del reporte generado por la herramienta donde se encuentran reportados los elementos señalados en la Figura 4.5.8.

Elementos que recibieron el foco mediante teclado

Mediante la detección de elementos que reciben el foco de teclado, también logramos re-validar los resultados del reporte de elementos con handlers JS.

Como se puede ver en la Figura 4.5.12, los elementos de tipo link <a> que se catalogaron como inaccesibles previamente en el reporte de elementos con handlers JS, tampoco han podido recibir el foco de teclado. Por lo tanto, en la sección de elementos interactivos también se les agregó la etiqueta de “no accesibles”. Por ejemplo, a los links “Ver Sucursales”, “Calcular costo de envío” y “Ver Más”.

Por otro lado, el link “Envío gratis en AMBA” es accesible y en el reporte se pueden visualizar la cantidad de veces que recibió el foco de teclado por parte del usuario.

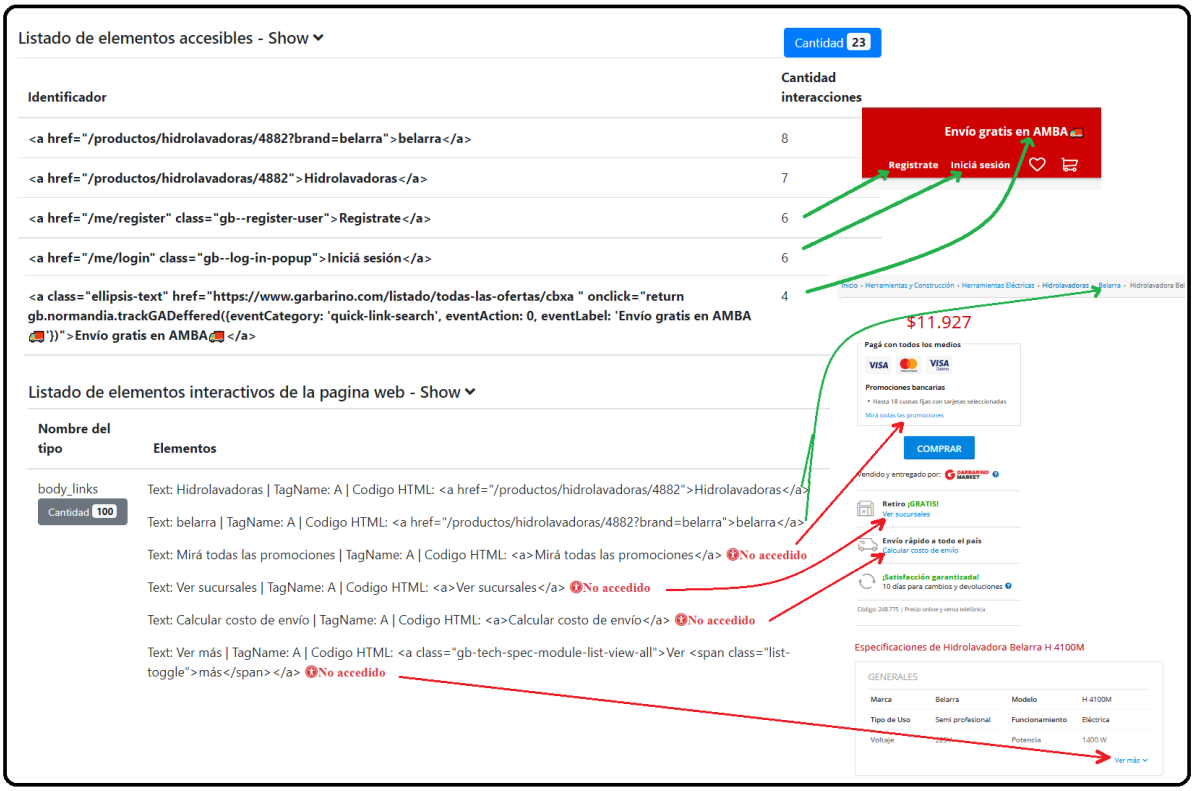


Figura 4.5.12: Captura del listado de elementos de tipo link que recibieron el foco durante la interacción de usuarios en la página.

4.5.3 Sitio web con elementos que poseen handlers JS #3

Blog de noticias Genius.com

URL

www.genius.com

Captura

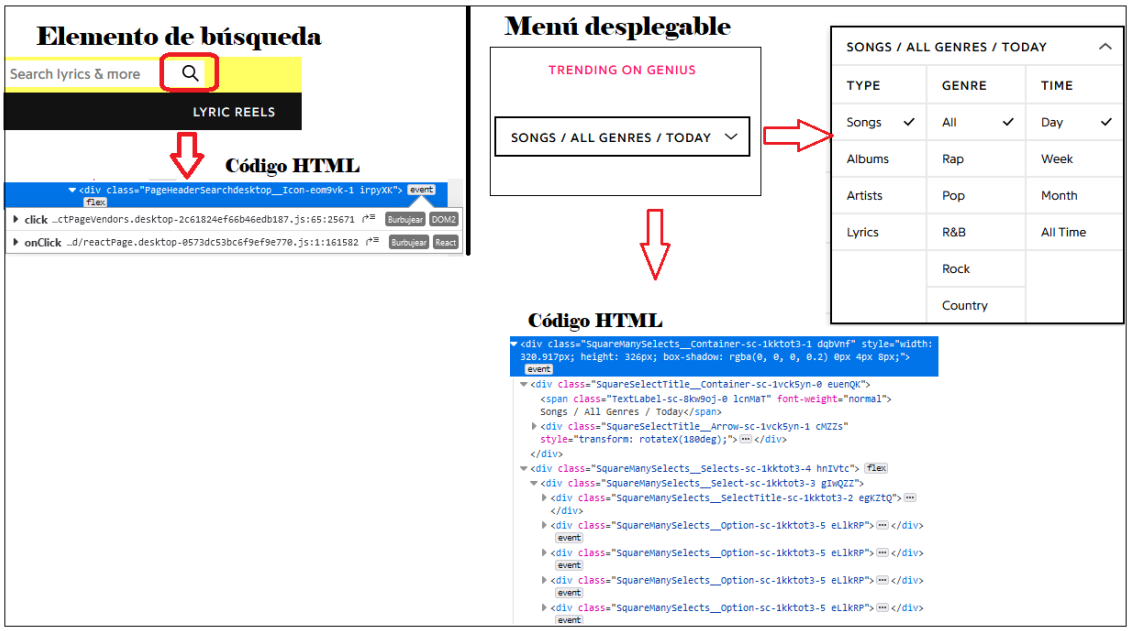


Figura 4.5.13: Diagrama que muestra cómo están declarados en el código un menú desplegable opciones de filtrado y el elemento de búsqueda interna en el sitio.

Comportamiento de NVDA

En la sección 4.3.2, se mencionaron dos elementos con forma de botones que parecían estar declarados de igual manera, pero esto no era así. Uno estaba declarado utilizando un y handler onclick, lo que no fue detectado por NVDA. Mientras que los dos botones restantes estaban declarados como links <a> con sus respectivos atributos href, estos sí fueron mencionados al usuario por NVDA.

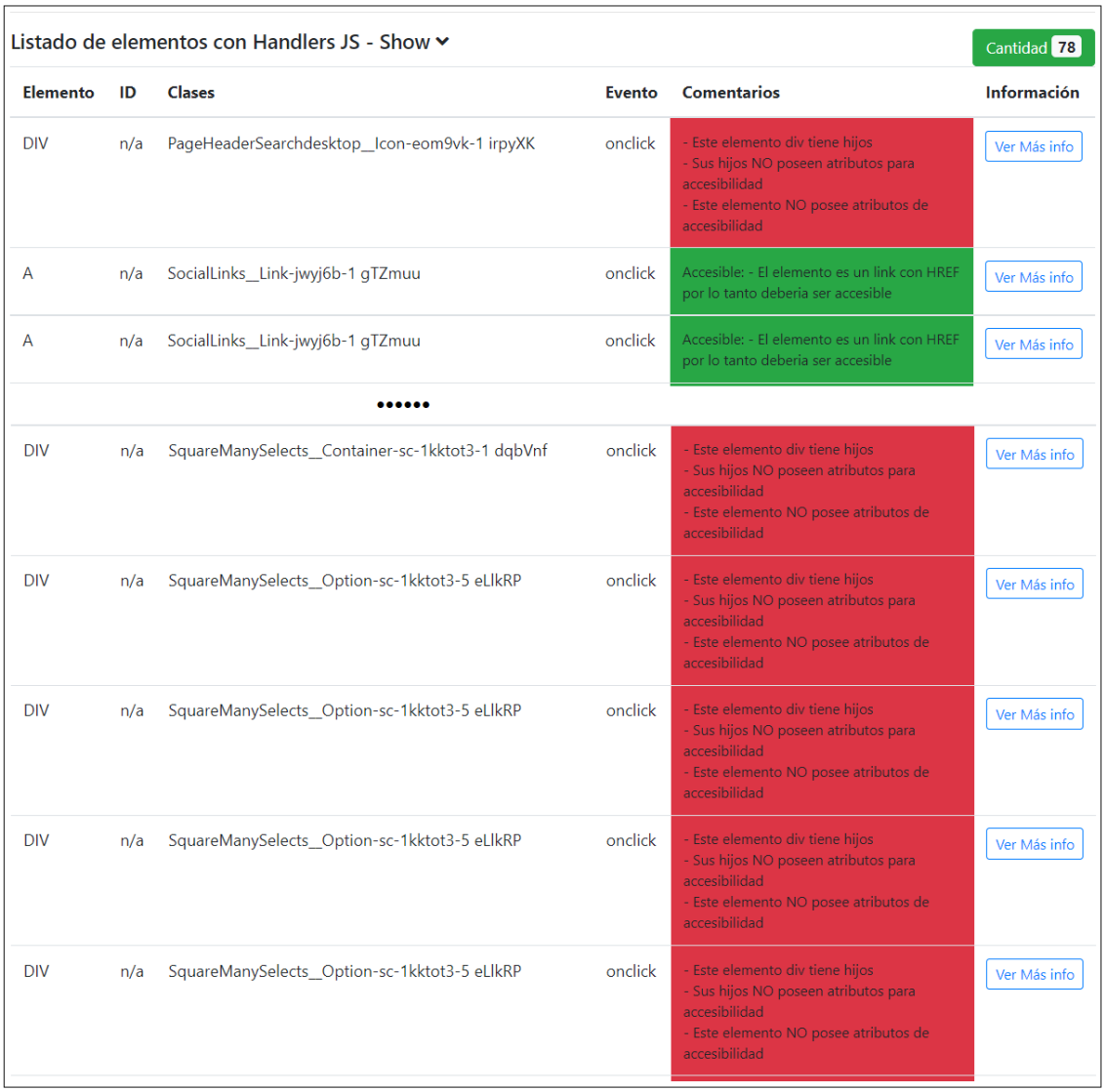
Los elementos señalados en la Figura 4.5.13, tampoco han sido mencionados por NVDA. Esto es debido a que están declarados con 's junto a un handler JS on click, sin utilizar ningún tipo de atributo para lograr accesibilidad.

Este es un caso muy interesante ya que posee una gran cantidad de elementos con handlers JS asignados, donde la gran mayoría son accesibles como lo muestra el reporte a continuación.

Reporte final de la herramienta

Como mencionamos anteriormente, el reporte generado por la herramienta es bastante extenso como para mostrarlo completamente. Esto se debe se listan en el reporte todos los elementos con handlers JS, accesibles como inaccesibles.

En la Figura 4.5.14, se puede observar que el elemento de búsqueda señalado en la Figura 4.5.13 fue reportado correctamente como elemento inaccesible. También el menú desplegable y sus elementos fueron reportados como inaccesibles.



Listado de elementos con Handlers JS - Show

Cantidad 78

Elemento	ID	Clases	Evento	Comentarios	Información
DIV	n/a	PageHeaderSearchdesktop__Icon-eom9vk-1 irpyXK	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info
A	n/a	SocialLinks__Link-jwyj6b-1 gTZmuu	onclick	Accesible: - El elemento es un link con HREF por lo tanto debería ser accesible	Ver Más info
A	n/a	SocialLinks__Link-jwyj6b-1 gTZmuu	onclick	Accesible: - El elemento es un link con HREF por lo tanto debería ser accesible	Ver Más info
.....					
DIV	n/a	SquareManySelects__Container-sc-1kktot3-1 dqbVnf	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info
DIV	n/a	SquareManySelects__Option-sc-1kktot3-5 eLlkRP	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info
DIV	n/a	SquareManySelects__Option-sc-1kktot3-5 eLlkRP	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info
DIV	n/a	SquareManySelects__Option-sc-1kktot3-5 eLlkRP	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info
DIV	n/a	SquareManySelects__Option-sc-1kktot3-5 eLlkRP	onclick	- Este elemento div tiene hijos - Sus hijos NO poseen atributos para accesibilidad - Este elemento NO posee atributos de accesibilidad	Ver Más info

Figura 4.5.14: Screenshot parcial del reporte generado.

Mientras que los botones de la Figura 4.3.2, en el reporte aparecieron los tres elementos. Esto es debido a que los tres elementos poseen handlers JS asignados, pero solamente el que está declarado utilizando un fue catalogado como inaccesible. Este elemento no posee hijos accesibles ni atributos para lograr proveer accesibilidad, mientras que los otros dos elementos declarados con <a> si se muestran como accesibles. Esto es correcto ya que se validó con NVDA que estos elementos son accesibles.

Elementos que recibieron el foco mediante teclado

La Figura 4.5.14, muestra que los botones diseñados con links de la Figura 4.3.2 y catalogados como accesibles también recibieron el foco de teclado.

Figura 4.5.14: Captura del listado de elementos de tipo link que recibieron el foco durante la interacción de usuarios en la página.

4.5.4 Ampliación de búsqueda para elementos no detectados

Luego de varias pruebas en diferentes sitios web, el código del finder implementado para la problemática planteada en este capítulo fue recibiendo varios ajustes. Esto logró que la eficacia de la detección sea muy buena y precisa, como quedó demostrado en los ejemplos citados anteriormente.

Por ejemplo, luego de testear la herramienta en varios sitios web, se han logrado detectar la gran mayoría de elementos con handlers JS. Aun así, había algunos elementos que no podían ser detectados, y la causa fue que utilizaban handlers diferentes a los que eran buscados por el finder de búsqueda. Por ejemplo, en un principio el finder buscaba elementos con handlers "*onClick*", pero esto no detectaba la mayoría ya que existen elementos que manejan el evento "*click*". Por lo tanto, luego de añadir este evento a la búsqueda del finder, se logró ampliar aún más la cantidad de elementos detectados por el finder. Por ejemplo:

En el sitio de autogestión de APR, hay elementos de tipo link <a> los cuales antes de incluir eventos "click" en el finder, no podían ser detectados y ahora lo son. Como se puede observar en la Figura 4.5.16, se pueden ver los elementos con handlers JS accesibles y los que no lo son.

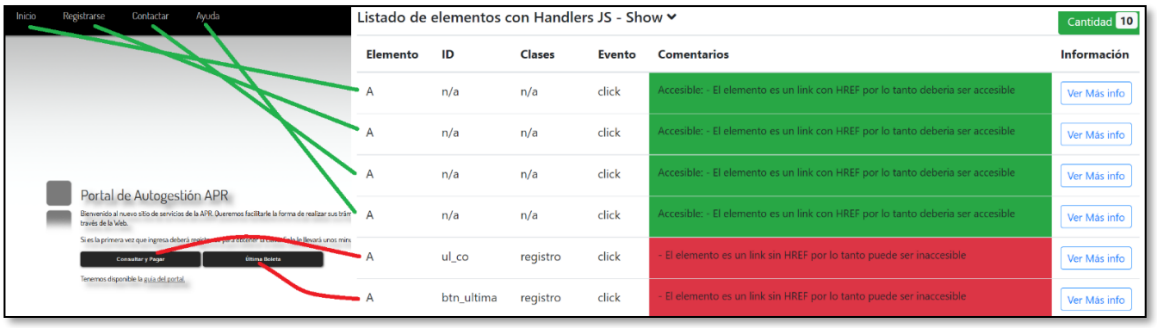


Figura 4.5.16: Diagrama que muestra los elementos con handlers JS de la página de APR reportados y clasificados.

4.5.5 Eventos no detectados

Para seguir ampliando la capacidad de detección de este finder, se necesita seguir añadiendo más tipos de eventos a la etapa de búsqueda del algoritmo.

Para esta tesina nos enfocamos en elementos que reaccionan a eventos **onClick** y **click**, debemos destacar que para elementos de tipo link <a> no solo realizan su acción al recibir el click de mouse sino también al presionar la tecla *enter*. Pero al igual que los botones, esta es una característica de estos elementos. Para el caso de los y esto no es así, el usuario no los va a poder utilizar si no poseen una handler para recibir eventos de tecla *ENTER* (con handler *onclick* no es suficiente).

No obstante, hay casos donde se crean elementos interactivos que reaccionan a otro tipo de eventos, por ejemplo, como se puede ver en la Figura 4.517, los dos elementos para manipular el carrusel de imágenes están declarados con 's que tienen los siguientes handlers asignados: **mousedown**, **mouseup**, **touchend** y **touchstart**.

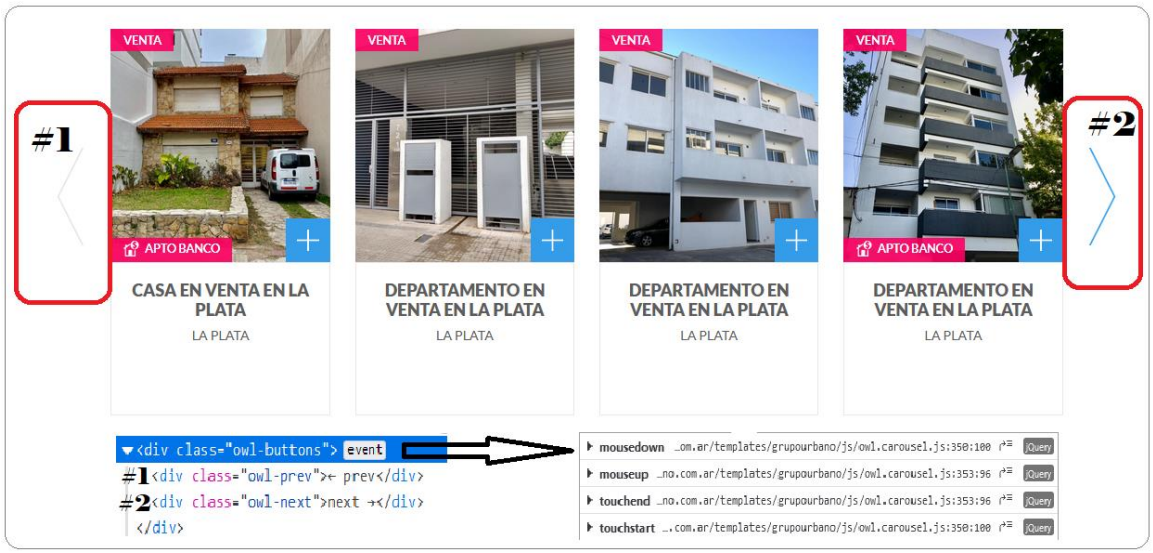


Figura 4.5.17: Captura del carrusel de imágenes del sitio web de grupo-urbano.com.ar, junto al código HTML y los handlers asignados.

Sería bueno para trabajos futuros incluir estos eventos a la búsqueda de elementos.

CAPÍTULO 5

ANÁLISIS DE ACCESIBILIDAD EN MENSAJES DINÁMICOS DE FORMULARIO WEB

5.1 Introducción

Los formularios se utilizan para todo tipo de efectos interactivos en la web. Los formularios permiten a los usuarios seleccionar y comprar los productos, rellenar encuestas y cuestionarios, inscribirse en cursos, búsqueda de información en la web, y una larga lista diferentes acciones.

Cuando hablamos de la accesibilidad de los formularios, generalmente nos referimos a que sean accesibles para las personas que utilizan lectores de pantalla. Las personas con otros tipos de la discapacidad normalmente resultan menos afectadas por los formularios defectuosos [AccWebForm]. Cabe señalar, sin embargo, que todos nos beneficiamos de un formulario bien organizado y fácil de utilizar, especialmente aquellos con discapacidades cognitivas.

Asistencia a los usuarios para prevenir y corregir errores

Los formularios y otros elementos de interacción pueden ser confusos o dificultar su uso para muchas personas y como resultado, ellas serán más propensas a cometer errores. Algunos ejemplos de ayuda para prevenir y corregir errores serían:

- Instrucciones descriptivas, mensajes de error y sugerencias de corrección.
- Ayuda contextual para funcionalidades e interacciones más complejas.
- Opción de revisar, corregir o revertir envíos si es necesario.

El cumplimiento de este requisito ayuda a las personas que no ven o no oyen el contenido y que es posible que no reconozcan las relaciones implícitas, las secuencias y otros indicios visuales. También ayuda a las personas que no entienden la funcionalidad, están desorientadas o confundidas, olvidan o cometen errores usando formularios y elementos de interacción por cualquier otra razón.

Notificaciones de Usuario

Se debe proporcionar comentarios a los usuarios sobre los resultados del envío del formulario, ya sea exitoso o no [AccPrinc]. Esto incluye comentarios en línea o cerca de los campos del formulario y comentarios generales que generalmente se brindan después del envío del formulario.

Las notificaciones deben ser concisas y claras. En particular, los mensajes de error deben ser fáciles de entender y deben proporcionar instrucciones simples sobre cómo pueden resolverse. Los mensajes de éxito también son importantes para confirmar la finalización de la tarea [FormNotif].

5.2 Motivación del buen uso de notificaciones en formularios

Cuando se envía un formulario, es importante que se notifique al usuario si el envío se realizó correctamente o si se produjeron errores.

Cuando ocurren errores, es útil enumerarlos en la parte superior de la página, antes del formulario o al lado de cada campo del formulario [FormNotif].

Cada error mencionado debe:

- Hacer referencia al campo del formulario correspondiente, para ayudar al usuario a reconocer el input;
- Proporcionar una descripción concisa del error de una manera que sea fácil de entender para todos;
- Proporcionar una indicación de cómo corregir el error y recordar a los usuarios cualquier requisito de formato;
- Incluir un enlace dentro de la misma la página al campo de formulario correspondiente para facilitar el acceso a los usuarios.

Además de los consejos anteriores, cada mensaje de error debe tener el atributo role establecido en ***alert***, role=”alert”, para que los usuarios de screen reader estén al tanto de este cambio.

5.3 Soluciones de desarrollo para que los mensajes dinámicos sean accesibles

En esta sección describiremos las distintas maneras que podemos utilizar durante el desarrollo para lograr que los mensajes y alertas en formularios sean accesibles.

Utilizar atributos ARIA

Los desarrolladores pueden utilizar los *roles ARIA* y los *atributos aria-** para cambiar el significado expuesto (semántica) de los elementos HTML, de acuerdo con los requisitos descritos en WAI-ARIA, excepto cuando estos entren en conflicto con la semántica nativa o sean iguales al ARIA implícito de un elemento HTML dado.

Como describimos en el capítulo 2, estos atributos están destinados para garantizar accesibilidad y así evitar que los desarrolladores hagan que los productos de tecnología de asistencia (screen readers) informen información de la aplicación web (IU) sin sentido que no representa la IU real del documento.

Ejemplo:

```
<div class="text">
    <label id="tp1-label" for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre" size="20"
          aria-labelledby="tp1-label"
          aria-describedby="tp1"
          aria-required="true" />
    <div id="tp1" class="tooltip"
        role="tooltip"
        aria-hidden="true">El nombre es obligatorio</div>
</div>
```

Roles WAI-ARIA

Uno de los roles recomendados por WAI-ARIA para los mensajes de error es el rol “Alert”, este se puede utilizar para decirle al usuario que un elemento de la página fue actualizado dinámicamente. Los screen-readers van a comenzar a leer instantáneamente el contenido actualizado cuando este rol está presente. Ejemplo:

```
<p role="alert" style="display: none;">El alert va a ser disparado cuando el elemento se vuelva visible.</p>
```

Atributo FOR para elementos Label

Si el mensaje de error está declarado dentro de un <label>, simplemente con asociar el label con el input mediante el atributo for es suficiente para que el lector de pantalla lea el error. El screen reader leerá el mensaje al focalizar en el elemento.

Ejemplo:

```
<label for="username">Error, complete con su Nombre</label>
<input type="text" id="username">
```

5.4 Formularios Web con mensajes de error inaccesibles

Descripción del problema

Como describimos en la introducción de este capítulo, Capitulo 5, los formularios se han convertido en elementos de suma importancia dentro de un sitio web. Para eso no solo deben poder ser accesibles mediante el uso de teclado, sino también estar organizados y que cualquier cambio dentro de ellos debe ser mencionado al usuario [WebAIM].

Los cambios más importantes dentro de un formulario web que deben ser mencionados al usuario son los mensajes de error. Si el usuario no se entera por qué el envío del formulario falló, nunca va a poder corregir esos errores y así no logrará continuar con el flujo de la aplicación web. Por ejemplo, en la Figura 5.4.1 vemos el formulario de registro de Mercado Libre; si el usuario no logra percibir los errores marcados en rojo, nunca logrará crear un usuario para utilizar la aplicación.

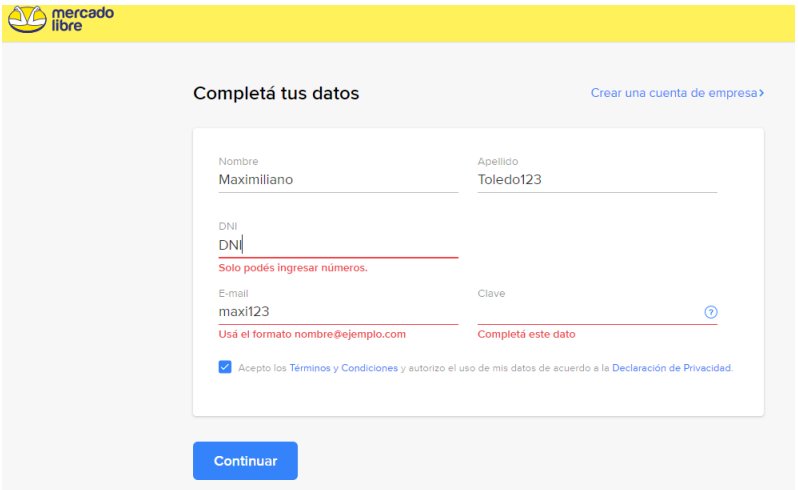


Figura 5.4.1: Ejemplo de errores de formulario desplegados, obtenido de la página para registrarse de Mercado Libre.

Ejemplos de formularios con mensajes dinámicos inaccesibles

A continuación, se describirán varios ejemplos de formularios de diferentes sitios web donde los mensajes de errores son inaccesibles. Estos casos fueron descubiertos en base al testeo mediante uso del screen reader *NVDA* en estos formularios web. Se notó que al desplegarse un mensaje de error o de *success* dinámico el screen reader no logró detectarlos y por ende no los mencionó.

Ejemplo #1: Portal de Autogestión APR

Sitio Web: https://autogestion.apronline.gov.ar/

Formulario de login

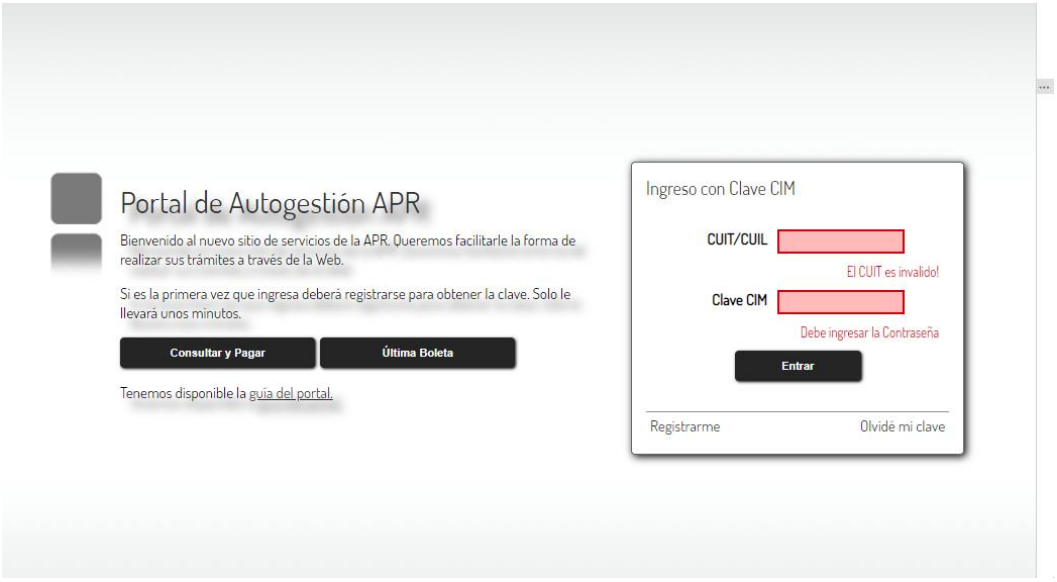


Figura 5.4.2: Formulario de Ingreso con varios mensajes de error desplegados.

Comportamiento en la navegación con el uso de screen reader

Si dejamos los campos del formulario vacíos, cuando presionamos en el botón “Entrar”, va a aparecer un error debajo de cada input. El foco se va a mover al input que contiene el error, NVDA nos dice “cuit campo de entrada”, pero no menciona el mensaje de error que describe el motivo por el cual falló el envío del formulario.

El problema de accesibilidad se genera porque el mensaje de error contiene el atributo *for*, pero está declarado como un elemento <*span*> y no como <*label*>. Cabe destacar que el atributo *for* solo funciona en elementos *label* y *output*.

Ejemplo #2 Formulario para crear cuenta Garbarino Online

Sitio web: https://www.garbarino.com/

Formulario de Registro

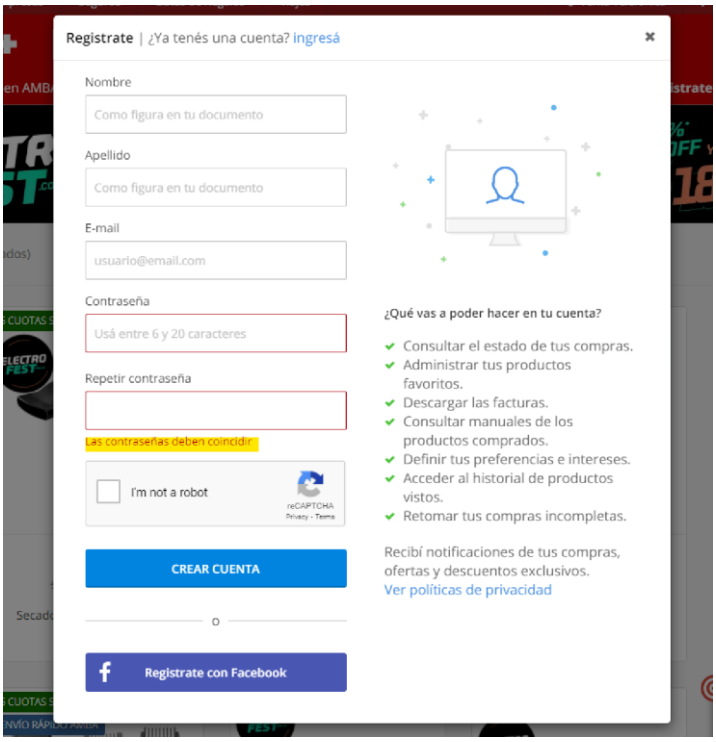


Figura 5.4.3: Formulario de Registro con varios mensajes de error desplegados.

Comportamiento en la navegación con el uso de screen reader

Cuando se deja algún campo vacío y se presiona en “Crear Cuenta”, se despliega un mensaje de error y *NVDA* nunca se entera. Ni siquiera hay foco al *input* que generó el error. El elemento que contiene el error no posee ningún tipo de atributo *ARIA* que ayude a *NVDA* a capturarlo y así mencionárselo al usuario.

Ejemplo #3 Formulario de creación de cuenta diario La Nación

Sitio web: https://ingresar.lanacion.com.ar

Formulario para Registrarse

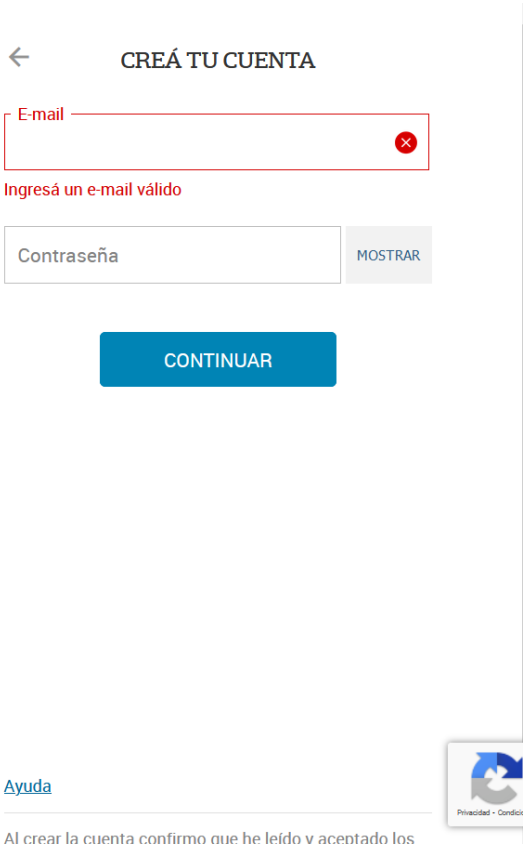


Figura 5.4.4: Formulario de Registro con varios mensajes de error desplegados.

Comportamiento en la navegación con el uso de screen reader

Cuando se presiona el botón “Continuar” y hay algún error en el formulario, se despliega un mensaje, pero NVDA nunca se entera. Ni siquiera hay foco al input que generó el error. Además, el foco queda posicionado en el mismo botón *submit* del *form*, por lo tanto, el contexto que se le presenta al usuario es muy confuso ya que parecería que el envío del formulario no funciona.

El mensaje de error está dentro de un elemento <*span*> sin ningún tipo de atributo o rol *ARIA* para hacerlo accesible como se observa en la Figura 5.4.5.

Figura 5.4.5: Código HTML del elemento *SPAN* que contiene el mensaje de error.

Ejemplo #4 Formulario de envío de pedido o consulta a Farmadelivery

Sitio web: https://farmalivery.com/contacto/

Formulario de envío de pedido o consulta

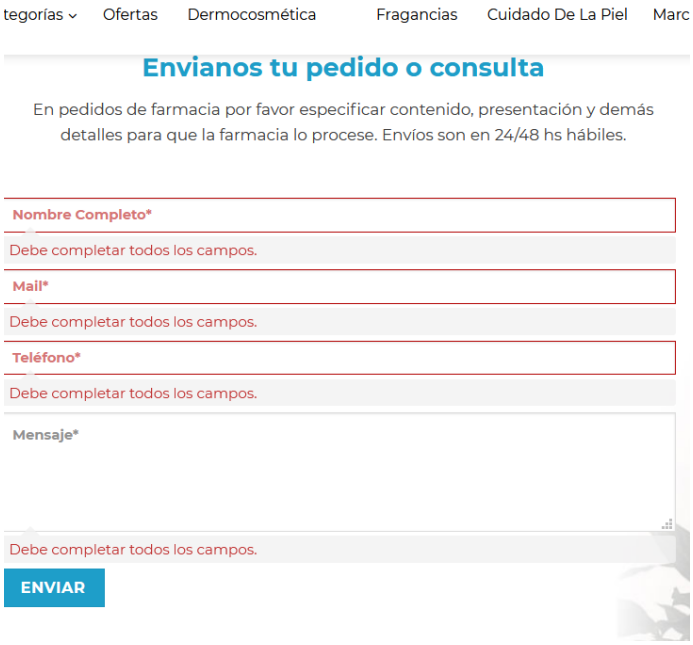


Figura 5.4.6: Formulario de Consultas con varios mensajes de error desplegados.

Comportamiento en la navegación con el uso de screen reader

Cuando se presiona el botón “Enviar” y este contiene errores, se despliegan los mensajes de error, pero aun así *NVDA* nunca se entera sobre estos eventos y no los menciona al usuario. Ni siquiera hay foco al input que generó el error. El foco queda en el mismo botón *submit* “Enviar” del formulario.

Este caso es distinto a los anteriores ya que demuestra el mal uso de los atributos que nos provee WAI-ARIA para garantizar accesibilidad en los elementos de una página web [WAIARIA]. Como podemos observar en la Figura 5.4.7, los elementos *SPAN* y *DIV* que contienen los mensajes de error tienen el atributo *aria-hidden* con el valor *true*. Esto quiere decir que los screen readers tienen que ignorar a este elemento, por esto mismo, en este caso NVDA ignora los mensajes desplegados y por más que el usuario interactúe dentro del formulario, NVDA va a seguir ignorando y no va a leer los mensajes.

Esto es un caso típico de mal uso de atributos ARIA, ya que su principal propósito es el de garantizar accesibilidad. El atributo *aria-hidden* se utiliza generalmente para componentes web que son puramente para dar formato a la página y no incluyen contenido de utilidad para el usuario.

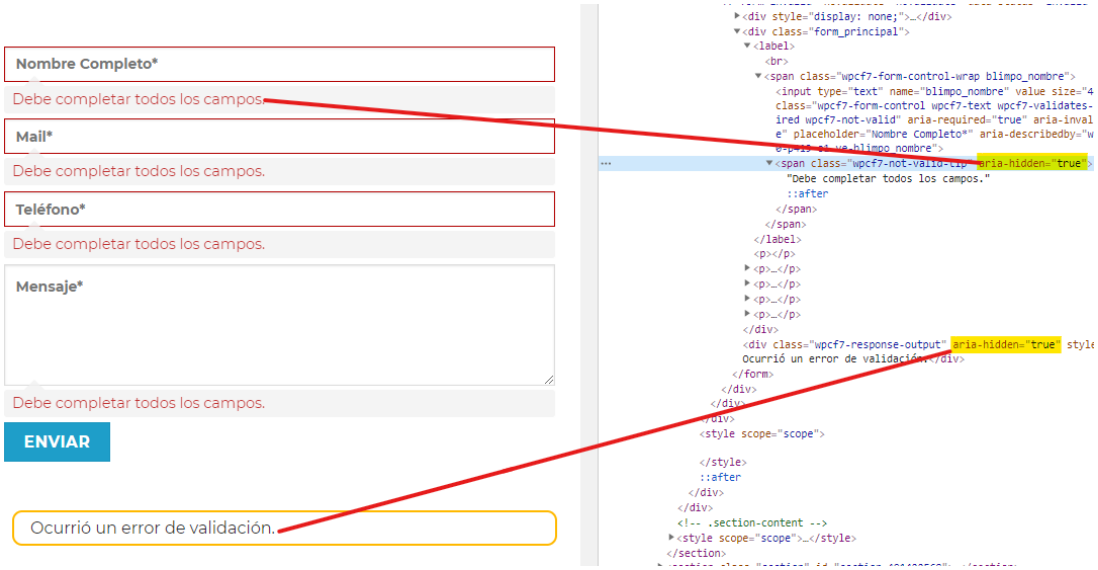


Figura 5.4.7: Código HTML de los elementos *SPAN* y *DIV* que contienen los mensajes dinámicos de error del formulario.

Ejemplo #5 Formulario de postulación para empleos del Hospital Italiano

Sitio web:

https://hiringroom.com/jobs/get_vacancy/59dce07203c26415d74fbcc1/candidates/new

Formulario de postulación para Hospital Italiano

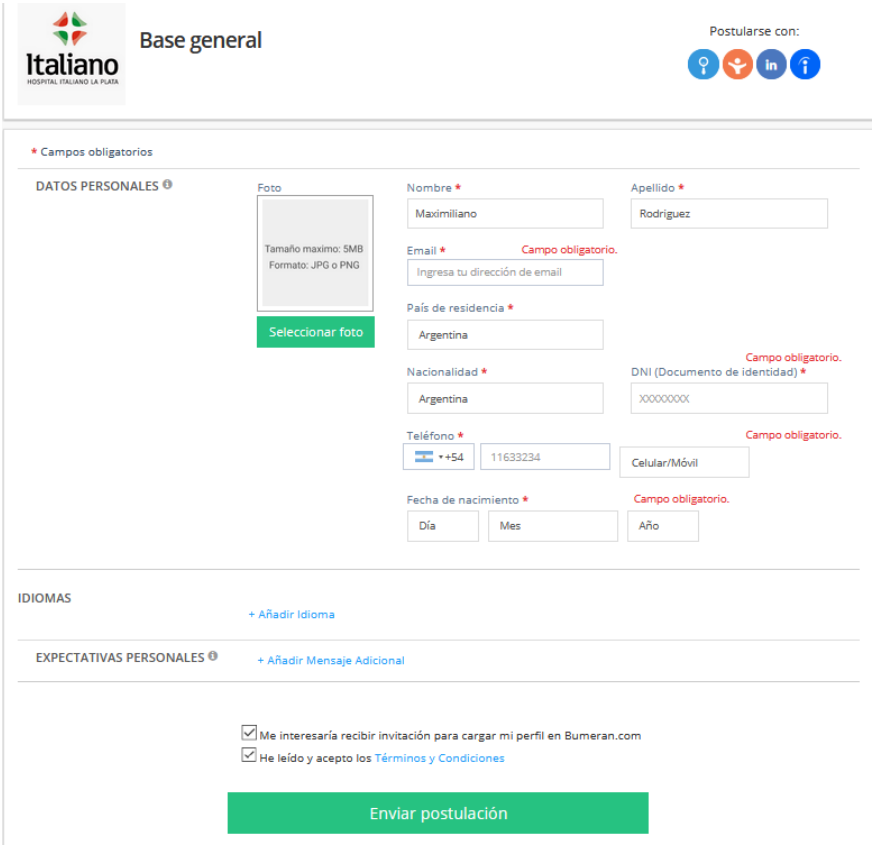


Figura 5.4.8: Formulario con varios mensajes de error desplegados.

Comportamiento en la navegación con el uso de screen reader

En ese ejemplo, NVDA no lee los mensajes de errores al presionar “Enviar postulación”. Además, se queda posicionado en el mismo botón *submit* sin darle ningún contexto al usuario.

Revisando el código fuente encontramos que estos mensajes están por fuera de un elemento *label*, declarados como un elemento *<p>* los cuales tampoco poseen ningún atributo *ARIA* para lograr ser accesibles.

Para que *NVDA* detecte y lea los mensajes, estos mismos deben estar como contenido del mismo *label* y no dentro de un elemento hijo del mismo.

Para que se entienda mejor, a continuación, se menciona cómo están declarados actualmente y como deberían estar para poder ser accesibles.

Mensaje de error inaccesible

```
<label class="font-blue-dark" for="txtEmail">Email<span>*</span></label>
<input type="email" id="txtEmail"  class="form-control" value="" placeholder="Ingresa tu dirección de email"
required="true">
<p class="error text-error txtEmail_error">Campo obligatorio.</p>
```

Mensaje de error accesible

```
<input type="email" id="txtEmail" class="form-control" value="" placeholder="Ingresa tu dirección de email"
required="true">
<label class="error text-error txtEmail_error" for="txtEmail" role="alert">Campo obligatorio.</label>
```

Figura 5.4.9: Diagrama donde se muestra como está definido actualmente el código *HTML* del input *EMAIL* y su correspondiente mensaje de error. Junto a cómo debería definirse el código para que sea accesible.

Tal cual se define en la Figura 5.4.9, una opción de desarrollo para que el mensaje de error sea accesible se basa en que este debería tener un *rol alert* y el atributo *for* para asociarlo al *input* de *Email* del formulario.

Ejemplo #6 Formulario de Login de Taringa

Sitio web: https://www.taringa.net/login

Login de Taringa

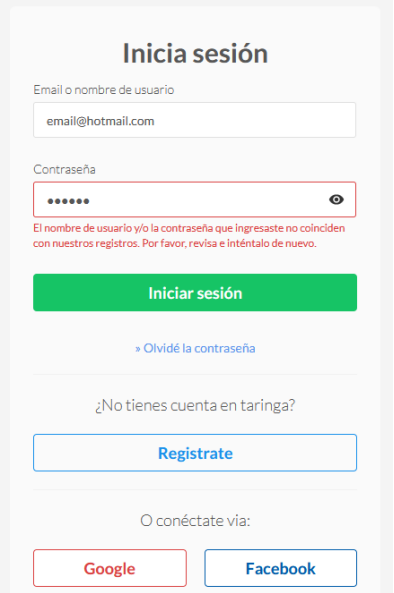


Figura 5.4.10: Formulario con mensaje de error desplegado en el login de Taringa.

Comportamiento en la navegación con el uso de screen reader

Cuando se presiona el botón “Iniciar sesión” y hay errores en el formulario, estos se muestran, pero aun así *NVDA* nunca se entera. Ni siquiera hay foco al *input* que generó el error. El foco queda en el mismo botón *submit* del *form* como en los ejemplos anteriores, sin darle ningún contexto al usuario.

Figura 5.4.11: Código HTML de los elementos *HTML* que contienen el mensaje dinámico de error del formulario.

En la Figura 5.4.11, podemos observar que en el nodo div, el cual contiene al mensaje de error, tenemos un arreglo llamado *childNodes*, el cual contiene los dos elementos hijos del *div*. En los cuales ninguno de los dos elementos *label* y *span* poseen ningún atributo o rol ARIA para hacerlos accesibles.

5.5 Herramienta para la detección y reporte de mensajes dinámicos inaccesibles en formularios web

5.5.1 Introducción a la herramienta

En esta sección daremos una breve introducción a la herramienta desarrollada para esta tesina, la cual lo que busca es lograr detectar de forma automática mensajes dinámicos en formularios que no cumplan con las reglas básicas de desarrollo para que puedan ser accesibles.

La detección de estos mensajes no se puede hacer de forma totalmente estática revisando el código al entrar a la aplicación web, ya que estos mensajes van apareciendo de forma dinámica dentro o fuera del formulario. La forma de generarse es algo compleja y dinámica como para saber si son accesibles o no simplemente observando el código. La forma más efectiva es generarlos y analizarlos.

El desarrollo e implementación de esta herramienta tomó como punto de partida la investigación y herramienta realizada por Watanabe, para identificar Drop-down Menu Widgets utilizando *Mutation Observers* y *Visibility Changes* [Watanabe16], Figura 2.3.3.

5.5.2 Descripción del método de detección de la herramienta

El mecanismo del funcionamiento de esta herramienta consiste en que, se obtienen los formularios de la página (elementos <*form*>) y en cada uno se asigna un *listener JavaScript* (*MutationObserver*) para lograr capturar los eventos de cuándo se le agrega un elemento al formulario.

Los mensajes de errores en los formularios son generalmente elementos *HTML* de texto como <*p*> y <*span*> que se agregan al <*form*> cuando hay un error en un determinado input del mismo.

De esta forma, observamos dinámicamente cuando aparecen estos mensajes de error mientras el usuario interactúa con el formulario. Esta es una de las principales diferencias con el trabajo de desarrollado por Watanabe como lo mencionamos en el capítulo 3. Ya que, en su herramienta, las interacciones de usuarios son simuladas por un simulador el cual él denomino Web Robot. Pero la utilización de simuladores genera un decremento en la precisión por falsos positivos y el tiempo requerido para la simulación es muy alto [Watanabe16].

Durante una secuencia de interacción del usuario, lo esperado seria que, si el usuario comete algún error completando el formulario y el error se hace visible, el usuario de screen reader también sea notificado.

Si esto no sucede, se le generarían grandes dudas al usuario de cuál y porqué el formulario es invalido.

Al desplegarse estos errores, van a ser detectados como un cambio en el *DOM*. Estos pueden estar mezclados entre varios cambios dentro del formulario así que debemos filtrar por tipo de elemento *HTML*. Nos debemos quedar con los elementos de texto y luego analizarlos. En este caso analizaremos los elementos: <*p*> y <*span*>.

Luego de obtener estos elementos, analizamos si estos contienen los atributos necesarios para que sean leídos por los screen reader y así ser accesibles a los usuarios.

Pasos generales:

1. Validar que el elemento sea un *<p>* o **.
2. Validar que tenga contenido. Que no esté vacío ya que representa a un mensaje de error.
3. Validar los atributos del elemento, y que entre estos tenga atributos *ARIA* o *for* para elementos *label*.
4. Reportar si es inaccesible.

5.5.3 Implementación de la herramienta

1. Se utilizó una extensión web para implementar esta solución ya que mediante la *DOM API* que nos provee el mismo browser tenemos acceso a todos los elementos *HTML* de la página de una forma sencilla.
2. Obtenemos todos los formularios de la página web actual:

```
var formElements = document.querySelectorAll('form');
```

3. A cada elemento *<form>* le asignamos un observable para capturar los cambios en el *DOM* del elemento:
   a. Declaramos el observable y la función de *callback* que se va a ejecutar cuando ocurre un cambio en el *DOM*:

```
var mutationObserver = new MutationObserver(function(mutations) {
    mutations.forEach(function(mutation) {
      check_issues(mutation);
    });
  });
```

   b. Asignamos un observable a cada elemento *<form>*:

```
formElements.forEach(function (form) {
    mutationObserver.observe(form, {
        childList: true,
        subtree: true
      });
});
```

4. Cuando un cambio ocurre, filtramos por tipo de elemento. Solo nos quedamos con los elementos *p*, *span* y *label*.
5. Si no poseen atributos *rol* o *aria**, reportamos el elemento.
6. Se envía la información de este evento a la *API Rest* de la herramienta, y esta se encarga de darle el formato correcto para almacenar la información en la base de datos.
7. Dentro de la *API Rest*, hay un método que se encarga de realizar toda la lógica necesaria para lograr almacenar toda la información necesaria sobre los eventos de bad smells capturados por la extensión web. El método: *saveBadSmellEvent()*, dentro del mismo se almacena el evento de bad smell en la colección ***BadSmellEvents***, aquí se encuentran almacenados todos los eventos de bad smells. Sobre el evento se guarda:
   a. BaseURI: URL de la página web donde se originó el evento.
   b. Data: contiene información sobre atributos del elemento web
      i. Text: contenido del elemento.

ii. TagName: tipo de elemento, ej: P.
iii. OuterHTML: código HTML del elemento.

c. Type: Tipo de smell generado por el elemento, ej: FORM_MESSAGE
d. FechaCreacion: Fecha y hora en la que se generó el evento

8. Además, en este mismo método se registra el evento en otra colección llamada ***ReportedPages***. El propósito de esta es tener almacenados los elementos web que provocaron eventos de accesibilidad en cada página web en donde se ejecutó la herramienta de una manera más organizada. Cada elemento de esta colección lo denominamos *ReportedPage*, y contiene la siguiente información:
    a. baseURI: URL de la página web donde se ejecutó la web extensión.
    b. Reported_elements: contiene una colección de elementos que generaron smells de accesibilidad dentro de la página. De cada elemento almacenamos cierta información útil para lograr identificarlos y llevar un conteo de ocurrencias:
        i. TagName: tipo de elemento.
        ii. OuterHTML: código HTML del elemento.
        iii. Bad_smell_type: Tipo de bad smell generado.
        iv. Ocurrencias: cantidad de veces que se reportó este evento.
        v. UltimoUpdate: Fecha y hora en la que se actualizo por última vez.
    c. FechaCreacion: Fecha y hora en la que se generó el evento.
    d. UltimoUpdate: Fecha y hora en la que se actualizo por última vez.
9. El paso final es armar el reporte de los *Bad Smells* para cada sitio web en los cuales se ejecutó la extensión web. De esto la encargada es la *SPA* de reportes, la cual se comunica con la *API Rest* para que esta le brinde la información que se encuentra en la base de datos.

5.6 Resultado final de la herramienta para la detección de mensajes de formularios web inaccesibles

Para mostrar el resultado final de la herramienta para detectar mensajes dinámicos en formularios web inaccesibles, se ejecutó en varios sitios web, con los siguientes resultados.

5.6.1 Sitio web con formulario #1

Página de registro del portal de APR

URL

https://autogestion.apronline.gov.ar/registro

Formulario

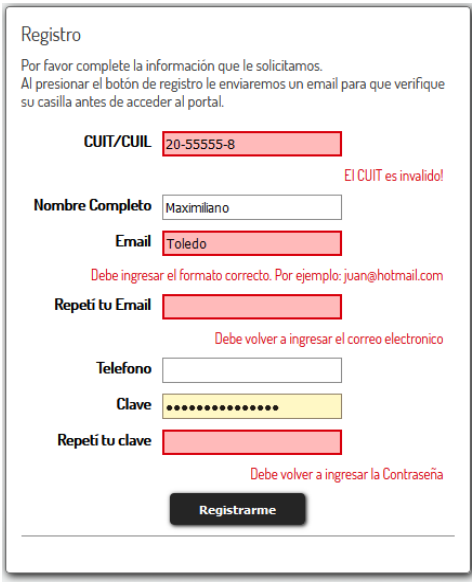


Figura 5.6.1: Formulario web del sitio web de APR con mensajes de error desplegados.

Comportamiento de NVDA

En este caso el screen reader *NVDA* no mencionó ninguno de los errores que se muestran en pantalla. Simplemente hizo un salto al primer input con error.

Reporte final de la herramienta

Reporta los elementos que representan a un error en el formulario y que no son accesibles.

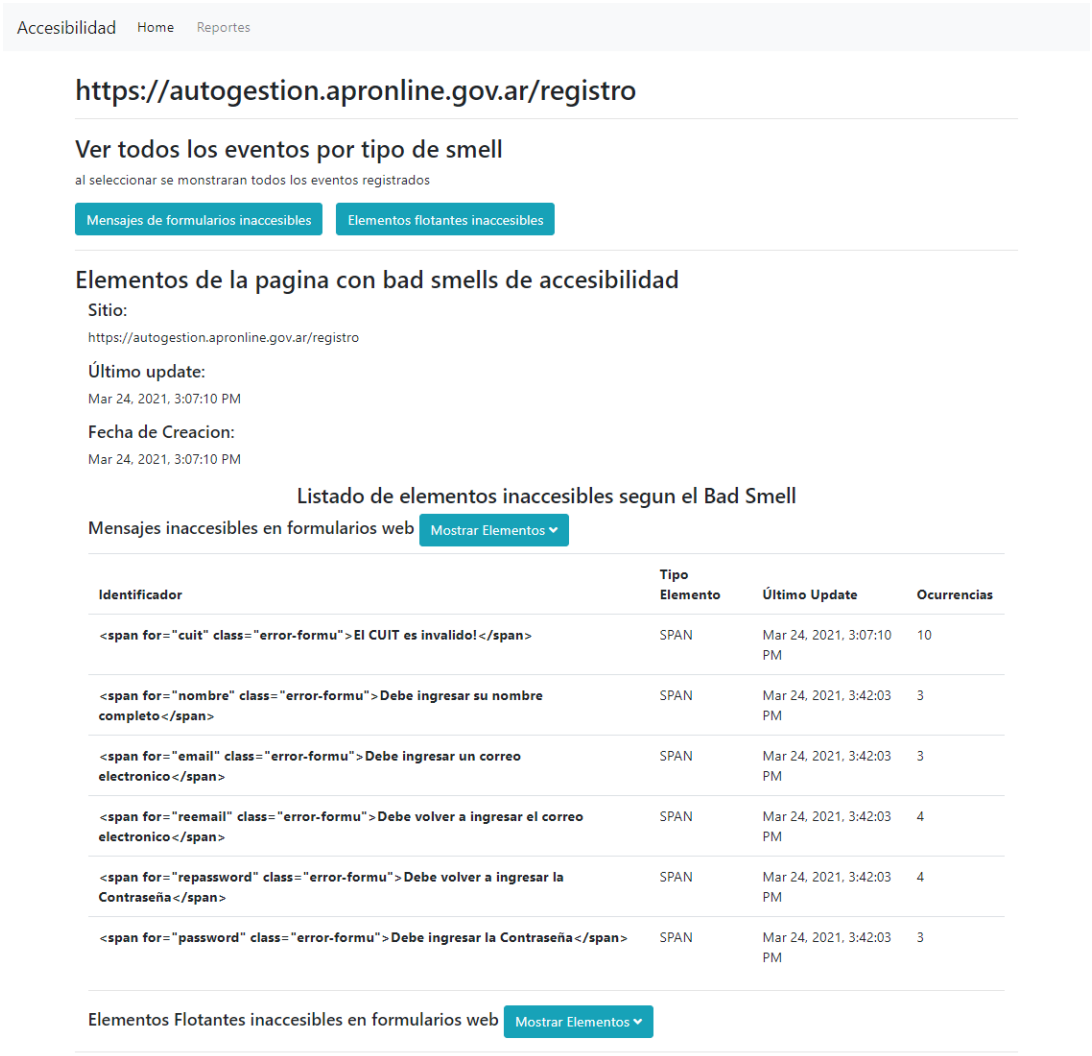


Figura 5.6.2: Reporte generado por la aplicación de reportes para la página de registro de APR.

En este ejemplo, los elementos que contienen los mensajes de error contienen el atributo for, pero están declarados como <*span*> y no como <*label*>. El atributo *for* solo funciona en elementos *label* y *output*, es decir en este caso no permite que el elemento sea accesible para el screen reader. Estos elementos fueron detectados y reportados por la herramienta, por lo tanto, podemos concluir que en este caso la detección funciono correctamente.

5.6.2 Sitio web con formulario #2

Página de postulación a empleos del Hospital Italiano

URL

https://hiringroom.com/jobs/get_vacancy/59dce07203c26415d74fbcc1/candidates/new

Formulario

Esta detección se realizó en el formulario web mostrado en la Figura 5.4.8.

Comportamiento de NVDA

En este caso el screen reader *NVDA* no mencionó ninguno de los errores que se muestran en pantalla. Ni siquiera se posiciono el foco en el *input* que genera el error, sino que el foco quedó posicionado en el mismo botón de "*Enviar postulación*".

El usuario no recibe ninguna ayuda ni comentario de lo sucedido. Generando así una posible desorientación al usuario sin entender si hubo un problema con algún campo erróneo del formulario o si el mismo no funciona y no envía solicitudes.

Reporte final de la herramienta

Reporta los elementos que representan a un error en el formulario y que no son accesibles.

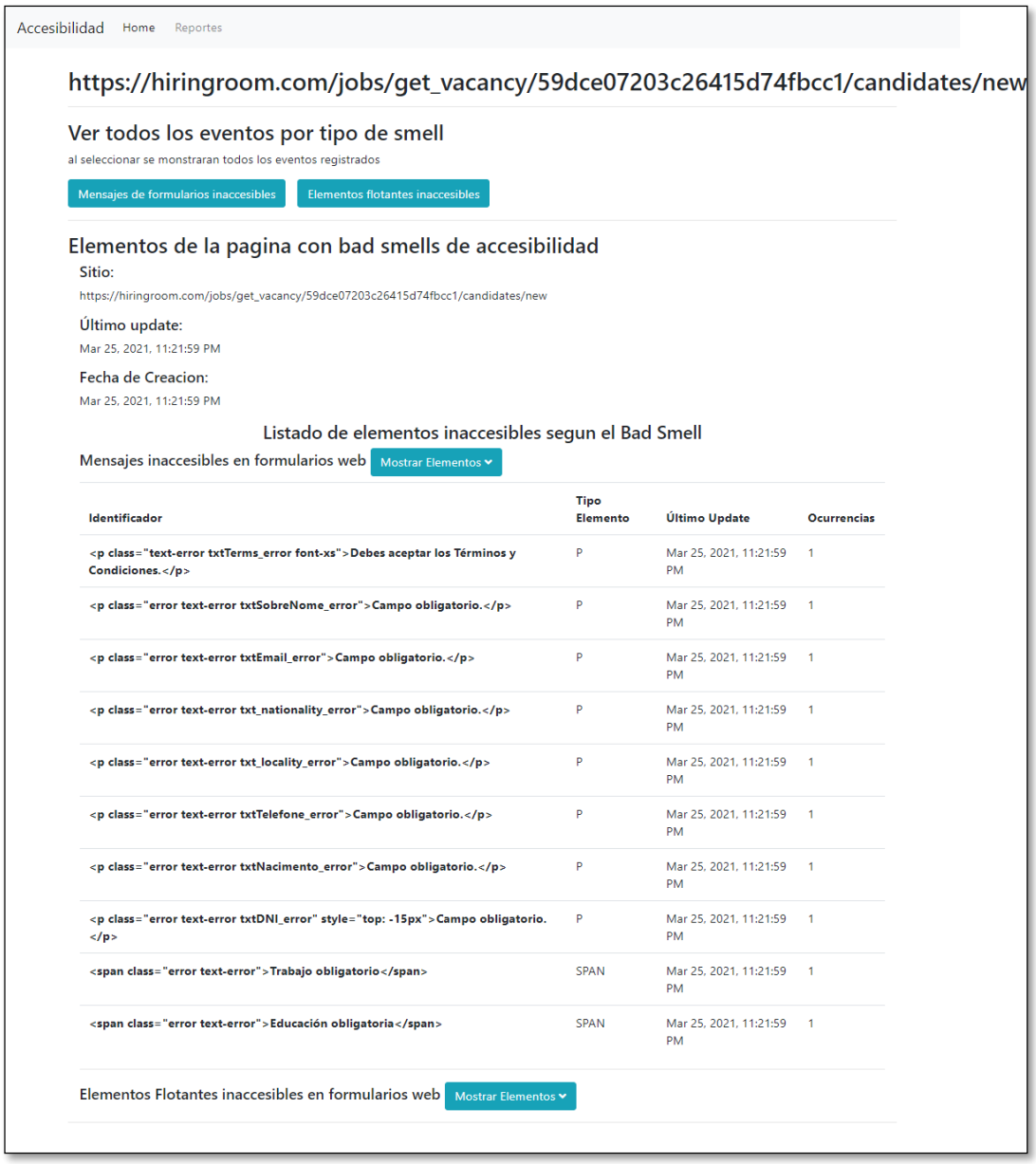


Figura 5.6.4: Reporte generado por la aplicación de reportes para la página de registro de APR.

En este ejemplo, también podemos decir que es un caso exitoso de detección para la herramienta ya que los elementos que contienen los diferentes mensajes de error en el formulario no poseen ningún tipo de atributo para ser accesibles y así leídos por *NVDA*.

5.6.3 Sitio web con formulario #3

Página de contacto de Farmacity

URL

https://www.farmacity.com/Farmacity/contacto

Formulario

Figura 5.6.5: Formulario web del sitio de contacto de Farmacity con mensajes de error desplegados.

Comportamiento de NVDA

Este ejemplo es diferente a los anteriores, ya que demostramos que la herramienta no genera falsos positivos cuando el formulario y sus mensajes de errores están desarrollados respetando los estándares de accesibilidad. En este caso, como es lo esperado, *NVDA* lee los mensajes al usuario cuando se despliegan si hay algún error al presionar “Siguiente”.

Reporte final de la herramienta

Como podemos observar el reporte esta generado, pero se encuentra vacío ya que en esta página no se detectaron bad smells de accesibilidad.

Figura 5.6.6: Reporte generado por la aplicación de reportes para la página de contacto de Farmacity.

El motivo por el cual los errores desplegados en el formulario son accesibles es porque poseen *Roles* y *atributos ARIA* para que cuando haya un error en el formulario, los screen readers lean el motivo de estos errores. Revisando el código del *form* los podremos apreciar:

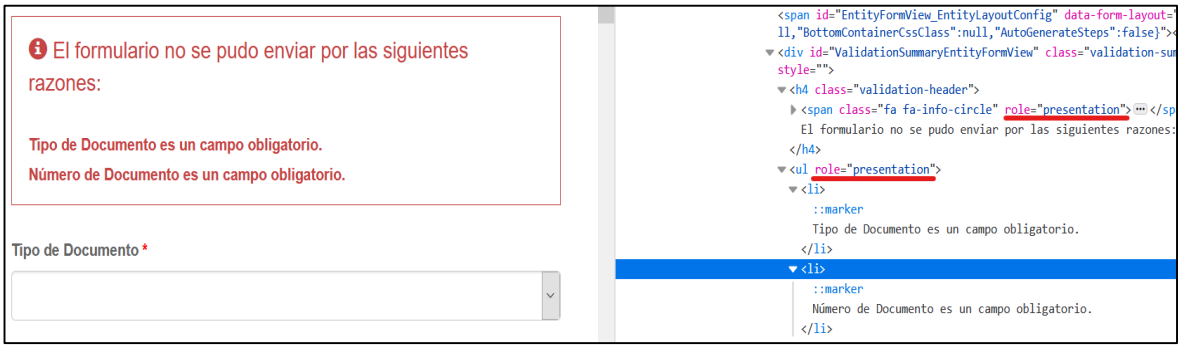


Figura 5.6.7: Código HTML del mensaje de error “Tipo de Documento es un campo obligatorio” en el cual vemos el uso de *Roles ARIA*.

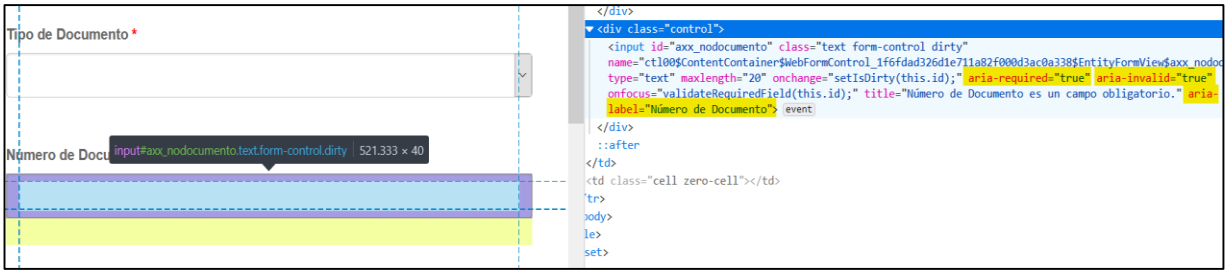


Figura 5.6.8: Código HTML del input que genera el error de “Número de Documento es un campo obligatorio” y podemos observar el uso de varios tipos de atributos ARIA para lograr accesibilidad en el elemento.

5.6.4 Sitio web con formulario #4

Página de registro de usuario para OpenFarma

URL

https://www.openfarma.com.ar/signup

Formulario

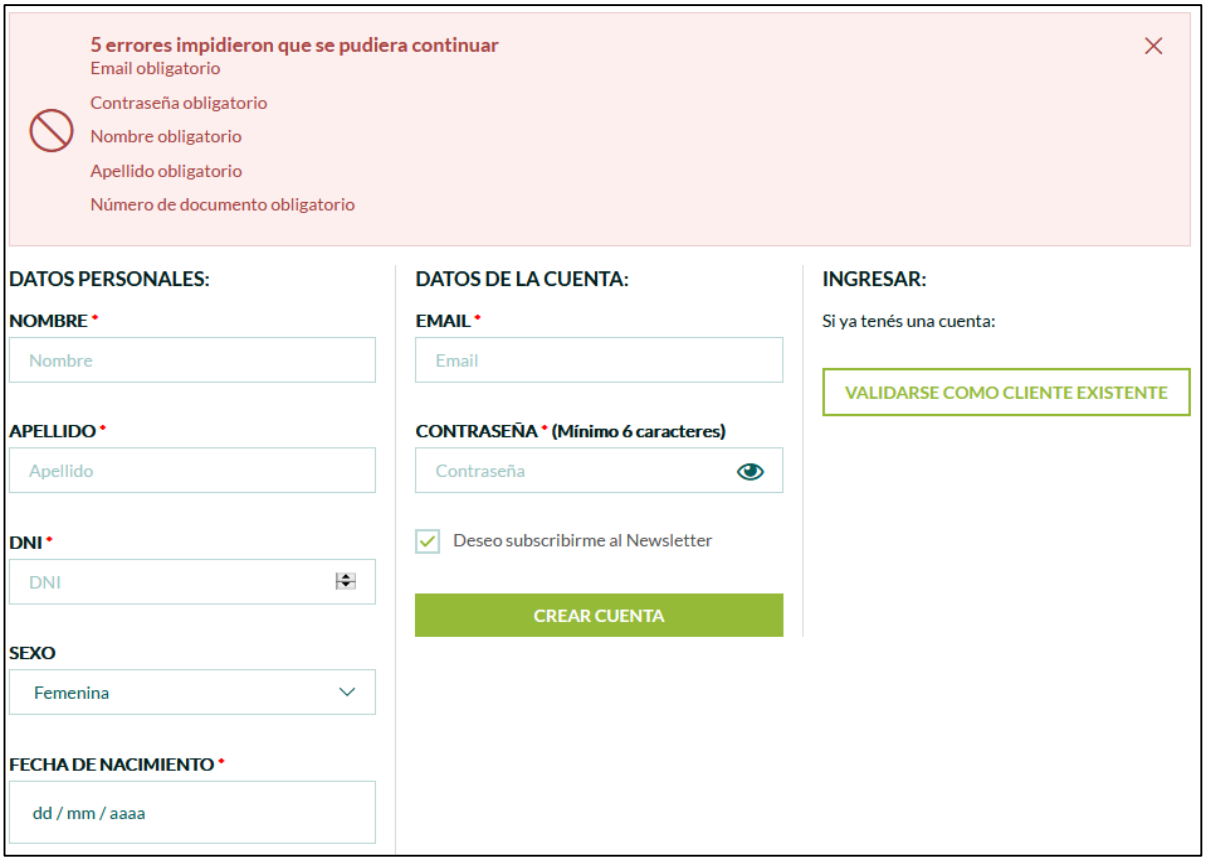


Figura 5.6.9: Formulario web del sitio de Creación de Cuenta de OpenFarma con mensajes de error desplegados.

Comportamiento de NVDA

En este ejemplo, cuando se generan los errores en el formulario, *NVDA* nunca se entera y no los menciona, ni siquiera son accesibles mediante el uso de la tecla TAB ni accesos directos de *NVDA*.

Reporte final de la herramienta

Figura 5.6.10: Reporte generado por la aplicación de reportes para la página de contacto de OpenFarma.

Como podemos observar el reporte esta generado, pero se encuentra vacío.

El motivo por el cual no se detectaron los elementos que generan los smells de accesibilidad es porque los mensajes de error del formulario son generados fuera del mismo, y como la herramienta está desarrollada para detectar solo cambios del DOM dentro del formulario no detecta a los mensajes de error en este caso.

Para trabajo futuro quedaría lograr integrar a la herramienta existente, la lógica para poder capturar estos mensajes por fuera del formulario.

CAPÍTULO 6

ANÁLISIS DE ACCESIBILIDAD EN ELEMENTOS WEB FLOTANTES

6.1 Introducción

Los modales son interfaces bastante comunes en la Web. Los desarrolladores y diseñadores pueden darles diferentes nombres: *lightbox*, *modal window*, *dialog*, *overlay*… pero son básicamente lo mismo. La W3C denomina a estos elementos flotantes como **dialogs** o **modal dialogs**.

Por lo general, los modales son fragmentos de texto que aparecen dentro de la ventana principal de la página web, superponiéndose con otro contenido e indicándole al usuario una acción o dándole un recordatorio. Los modales deben usarse con moderación. Al igual que las alertas, interrumpen el flujo de lo que hace un usuario si no puede controlar la apertura de un modal. Si el usuario puede controlar la apertura y cierre del modal con botones, esa es una alternativa mucho mejor.

Los modales deben usarse cuando hay pasos que el usuario debe realizar antes de que se pueda completar la tarea. El uso de un modal en lugar de una página completa permite a los usuarios mantener el contexto de su tarea [UXM].

En la web, en la mayoría de los casos, los modales se utilizan para solicitar a los usuarios alguna información de entrada (por ejemplo, completar un formulario o enviar una respuesta a una pregunta), mostrar información textual adicional o una versión más grande de una imagen, etc.

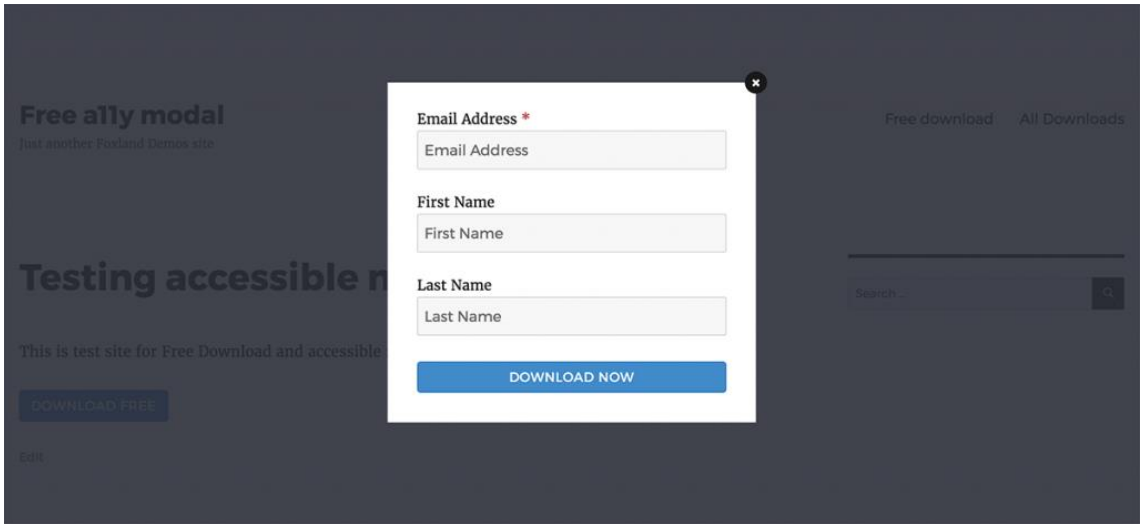


Figura 6.1.1: Ejemplo de un modal desplegado, en el cual el usuario solo puede interactuar con el contenido del mismo hasta que el modal se cierre.

Por lo general, *el resto de la página está atenuado y los usuarios no pueden interactuar con ella hasta que cierren el cuadro de diálogo*, como vemos en la Figura 6.1.1. En términos de ARIA, este es un diálogo "modal", **modal dialog**.

Sin embargo, también hay cuadros de diálogo "no modales", **non-modal dialogs**, que permiten la interacción con el resto de la página y en este caso se ven como "paneles" que flotan en la ventana principal. Estos paneles muestran algún contenido sobre la página web, mientras que el resto de la página continúa siendo 100% funcional y accesible, como puede verse en la Figura 6.1.2.

Figura 6.1.2: Esta captura de pantalla muestra un ejemplo de un elemento flotante “no modal”, presente en el sitio web de GPSFarma.

Durante nuestra búsqueda utilizando NVDA para detectar problemas de accesibilidad relacionados a elementos flotantes, descubrimos que hay una gran presencia de *non-modal dialogs* en varias las diferentes páginas web testeadas. En la sección 6.6 de resultados, hay ejemplos de sitios webs donde hay hasta más de 5 elementos non-modal dialogs presentes al ingresar a la página.

Luego de este análisis, en principio se decidió que un buen punto de partida era el de lograr poder detectar y analizar los non-modal dialogs presentes en un sitio web. Debido a que la detección de *modal dialogs* requiere de un proceso más complejo ya que son generados dinámicamente por acciones del usuario y muchos de ellos no están presentes al ingresar al sitio web.

En este capítulo, vamos a focalizarnos en los diálogos no modales, y vamos a describir cómo se incorporó la detección y análisis de este tipo de elementos, a la herramienta de detección de problemas de accesibilidad desarrollada para esta tesina.

En las siguientes secciones describiremos los diferentes tipos y daremos varios ejemplos de estos diálogos no modales presentes en diferentes sitios webs junto a sus problemáticas de accesibilidad.

6.2 Tipos y ejemplos de Non-modal dialogs

En la sección anterior describimos a los elementos *non-modal dialog* como aquellos pop-ups y paneles que cubren parte del contenido de la página web sin impedir el acceso al resto de la misma. Los diferentes tipos de *non-modal dialog* que podemos encontrar son:

6.2.1 Cookie notifications

El elemento es activado por la página después de que se carga (como las notificaciones de cookies, Figura 6.2.1) o después de cierto retraso (como las encuestas emergentes), es muy común y puede ser una molestia importante para los usuarios que utilizan el teclado para navegar. Algunos diálogos de tipo encuesta captan el foco, otros no. La mayoría de las notificaciones de cookies tampoco lo hacen, evitan cambiar el contexto de los usuarios de forma inesperada.

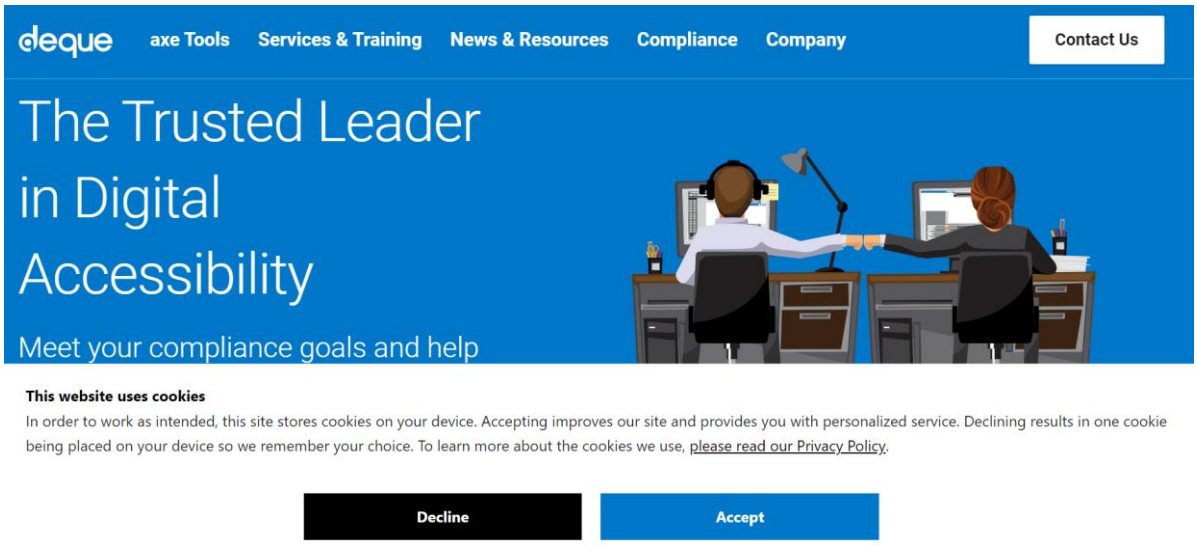


Figura 6.2.1: Ejemplo de pop-up que solicita el permiso del usuario para utilizar las cookies del browser. Sitio web: www.deque.com.

6.2.2 Toast messages

Existe una variante de mensajes generados por la misma página que desaparecen luego de unos segundos llamados "*toast messages*". Estos mensajes pueden simplemente decir "Su información ha sido guardada" o "El formulario ha sido enviado".

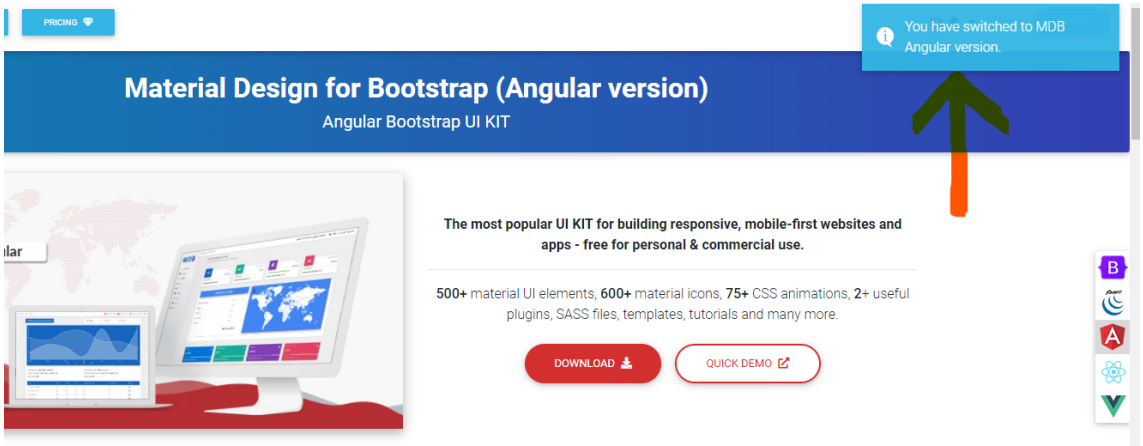


Figura 6.2.2: Ejemplo de un mensaje toast, el cual luego de 5 segundos desaparece. Sitio web: acc04.mdbootstrap.com.

6.2.3 Toggletips o Tooltips

Toggletips o mensajes de status aparecen durante la activación de botones y foco en determinados elementos, también se superponen al contenido de la página, pero no contienen elementos interactivos que deban enfocarse. Por lo tanto, se pueden leer a través de *aria-live* una vez que se muestran.

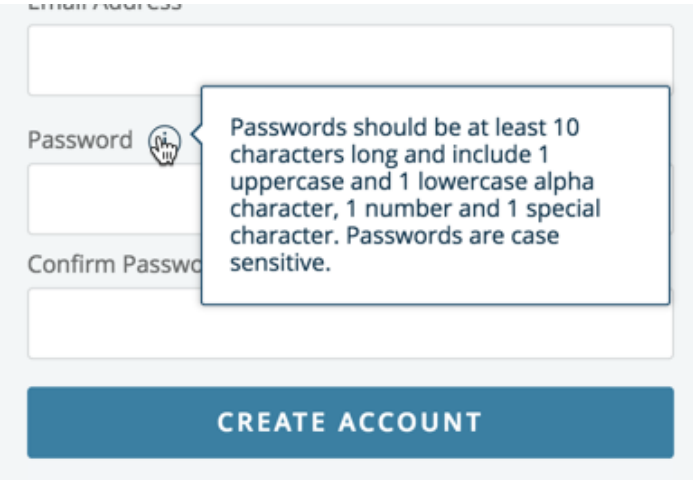


Figura 6.2.3: Ejemplo de un mensaje de tipo *Tooltip* o *ToggleTip*, el cual se muestra al posicionarse en el icono del input "*password*".

6.2.4 Popups activados por el usuario con elementos interactivos

Otro tipo de contenido son los *popups* que se activan mediante la ejecución de alguna acción disparadora, por ejemplo, al presionar algún otro elemento dentro de la misma página. Estos *popups* son bastante más complejos que los *Tooltips*, Figura 6.2.3. Ya que contienen elementos interactivos que el usuario puede operar para realizar varias acciones. El tamaño de estos elementos puede variar, pero pueden llegar a abarcar casi toda la pantalla; aun así, no capturan el foco y el usuario puede seguir navegando dentro del contenido de la página mientras el *popup* permanezca desplegado. En la Figura 6.2.4 y 6.2.5 se puede ver uno de los ejemplos más comunes de este tipo de elementos que son los asistentes virtuales.

Figura 6.2.4: Captura de la página web del gobierno de la Ciudad de Buenos Aires, la cual posee un asistente virtual el cual se activa presionando el diálogo "¿Cómo podemos ayudarte?" o el icono "BA".

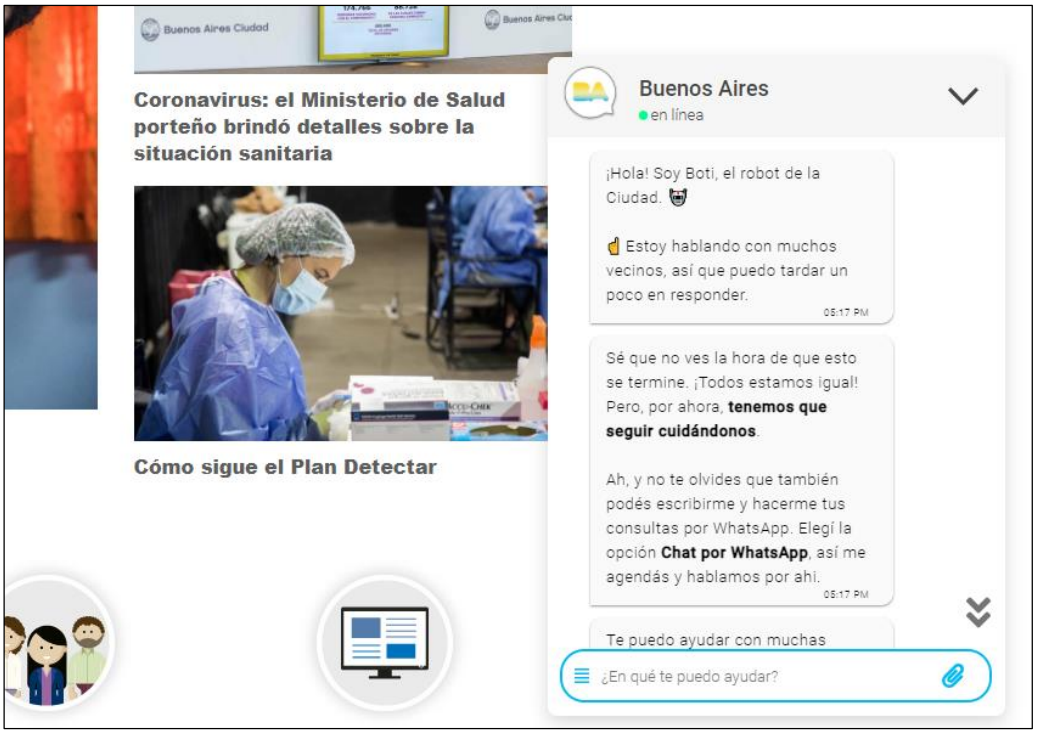


Figura 6.2.5: Asistente virtual que se despliega al presionar los elementos mencionados en la Figura 6.2.4

Como podemos observar en la Figura 6.2.5, el Asistente virtual contiene una gran cantidad de información que podría ser de gran utilidad para los usuarios, por ejemplo: pedir turno para vacunarse contra el COVID-19, teléfonos útiles y generación de denuncias (Violencia de género, Infancia, etc.), información sobre infracciones de tránsito, solicitud para hablar con un representante y muchas opciones más.

6.3 Problemas de accesibilidad en elementos flotantes

A continuación, se describirán mediante el uso de ejemplos, los diferentes problemas de accesibilidad que pueden tener los usuarios de screen readers respecto a los *non-modal dialogs*.

Estos casos fueron descubiertos en base al testeo mediante uso del screen reader *NVDA* en distintos sitios web que poseen este tipo de elementos.

6.3.1 Ejemplo #1

Diario El Dia, url: https://www.eldia.com/

Figura 6.3.1: Captura del sitio web del Diario El Dia.

En la Figura 6.3.1 podemos observar que hay un pop-up desplegado el cual es un modal simple para suscribirse a las notificaciones del diario, que es desplegado automáticamente al cargarse la página.

Al utilizar NVDA se pudo comprobar que el usuario solo va a poder darse cuenta de que existe ese elemento una vez que llegue al final de la página, solo luego de pasar por el último elemento va a poder interactuar con los botones del pop-up. Además de esto no se ofrece ningún tipo de contexto al usuario sobre el uso de los botones, solo se menciona el texto de cada uno de ellos, lo que seguramente logre confundir al usuario.

En la Figura 6.3.2 observamos que el código del pop-up implementado por un no contiene ningún atributo especial para lograr ser accesible.

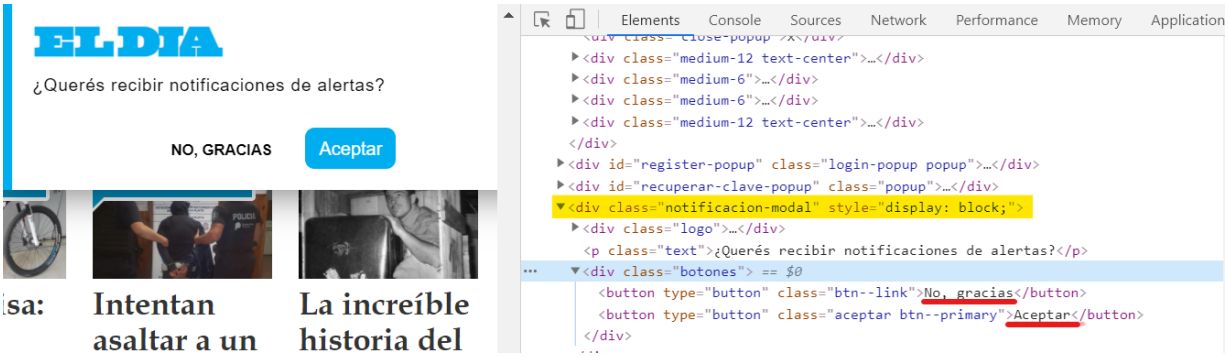


Figura 6.3.2: Código HTML del pop-up tomado como ejemplo del sitio web del Diario El Dia.

6.3.2 Ejemplo #2

GPS Farmacia, url: https://www.gpsfarma.com/

Figura 6.3.3: Captura de pantalla del sitio de GPS Farma.

En la Figura 6.3.3 podemos observar que hay un pop-up desplegado el cual es un modal simple para suscribirse a las notificaciones web de la página, el mismo es desplegado automáticamente al cargarse la página.

En este caso, al utilizar NVDA notamos que el usuario nunca puede acceder al pop-up. No se puede hacer foco (focus) en el mismo, por ende, no se puede acceder a ninguno de los links en su contenido. De hecho, el usuario de NVDA nunca se entera de que existe este elemento.

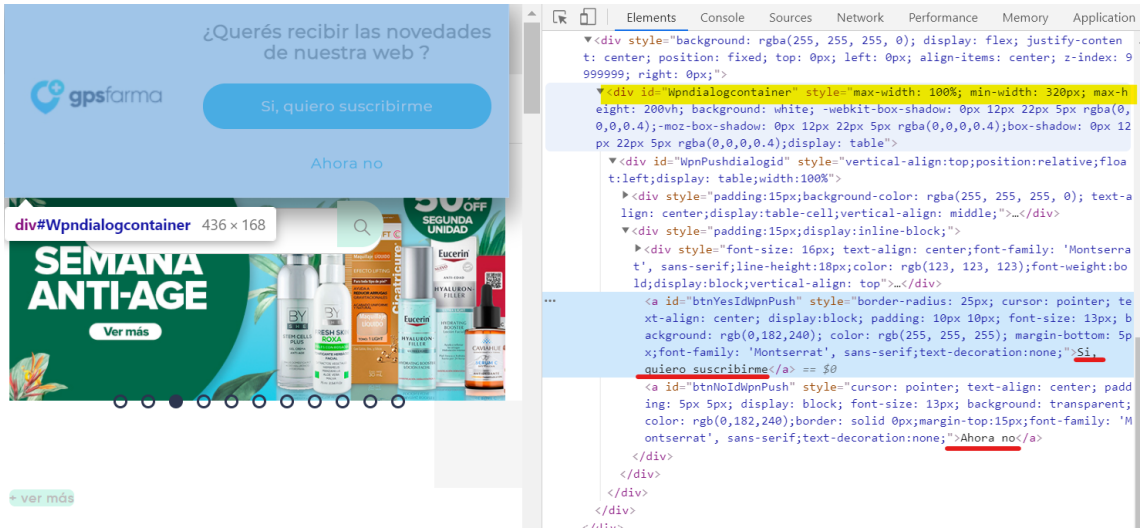


Figura 6.3.4: Código HTML del pop-up tomado como ejemplo del sitio web de la farmacia GPS.

Como observamos en la Figura 6.3.4, vemos que el pop-up es un flotante, pero sin ningún tipo de atributo o rol ARIA para lograr hacerlo accesible para personas que utilizan *screen readers*.

6.3.3 Ejemplo #3

Grupo Urbano Inmobiliaria, url: https://www.grupo-urbano.com.ar/

Figura 6.3.5: Elemento flotante activable que despliega el pop-up de WhatsApp.

Figura 6.3.6: Pop-up de WhatsApp desplegado.

Al utilizar NVDA en la página web de Grupo Urbano, se detectó que el elemento flotante de la Figura 6.3.5 no es detectado por el *screen reader*. Por lo tanto, el usuario nunca se entera de que este elemento se encuentra en la página. El elemento está presente desde que se carga la página, pero no se le provee al usuario una forma de poder acceder a él mediante teclado.

```
▼<div id="converse-chat-launcher"> == $0
    <div class="converse-no-seen">6</div>
  </div>
```

Código HTML del elemento web mencionado en la Figura 6.3.5

Al igual que en los ejemplos anteriores, este *non-modal dialog* está hecho por sin ningún atributo para hacerlo accesible a usuarios de screen reader.

6.3.4 Ejemplo #4

Diario Perfil, url: https://www.perfil.com/

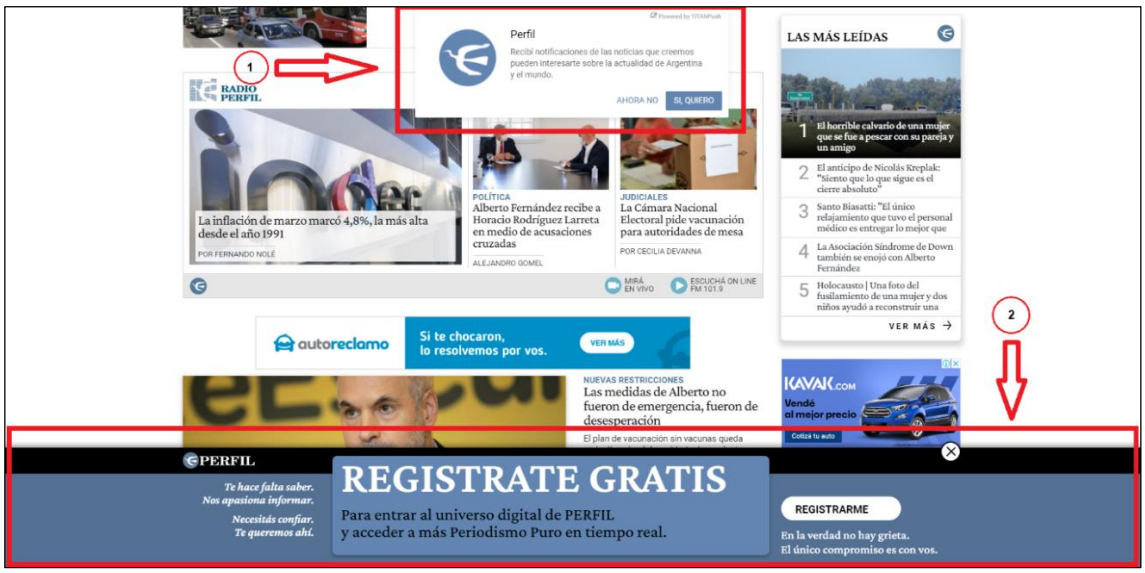


Figura 6.3.7: Captura del sitio web de Perfil.com donde están señalados y numerados los dos *non-modal dialogs* presentes en la página.

Este es otro caso de elementos con problemas de accesibilidad ya que ninguno de los dos fue detectado y mencionado al usuario por parte de NVDA.

Como observamos en la Figura 6.3.7 el sitio posee dos elementos flotantes, el elemento #1 sirve para que el usuario pueda recibir notificaciones web mientras que el #2 es para registrarse en el sitio web del diario. Se realizó un recorrido por toda la página, desde el principio hasta el final y en ningún momento NVDA pudo detectar que estaban presentes.

En las figuras 6.3.8 y 6.3.9 al analizar el código, observamos cual es el problema de que no sean accesibles y es porque no poseen ningún tipo de atributo o rol ARIA para poder serlo.

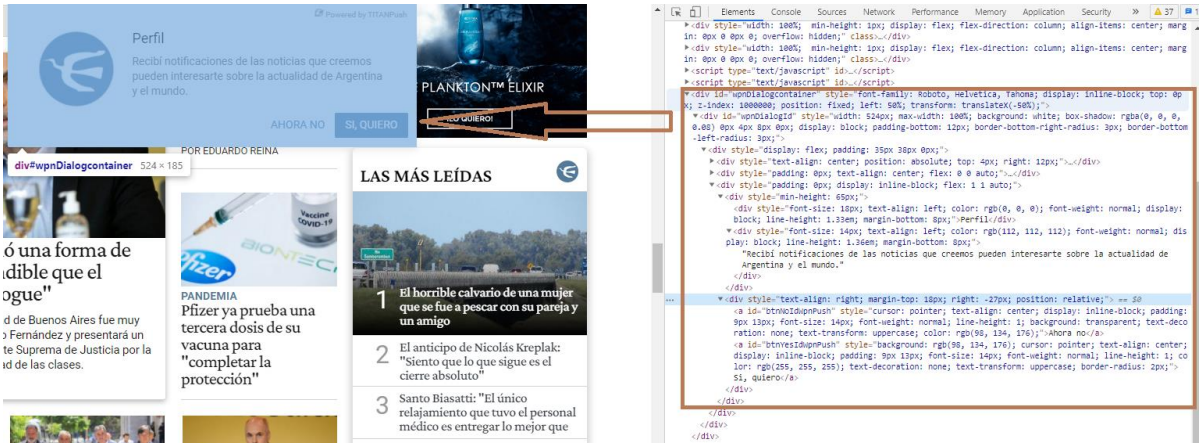


Figura 6.3.8: Código HTML del elemento #1, no posee ningún tipo de atributo ni rol ARIA para hacerlo accesible.

Figura 6.3.9: Código HTML del elemento #2, misma falta de atributos que en el elemento #1.

6.4 Consejos de accesibilidad para los non-modal dialog

6.4.1 Resumen

Ya hemos dado una breve introducción a los tipos de elementos modales que podemos encontrar en un sitio web actualmente. Además, mencionamos que en este capítulo vamos a focalizarnos en los *non-modal dialogs*.

Para tener una mejor comprensión de los bad accessibility smells presentes en estos elementos, en la sección 6.3 se describieron varios ejemplos en diferentes sitios web.

Antes de comenzar con la descripción de la herramienta para detectar estos elementos y sus problemas de accesibilidad, debemos comprender cuáles son las reglas básicas de desarrollo para conseguir que los *non-modal dialogs* sean elementos accesibles.

6.4.2 Tips generales para lograr accesibilidad en *non-modal dialogs*

Esta sección se divide en dos partes, en la primera describiremos tips generales para proveer accesibilidad en los *non-modal dialogs* según el sitio Access&Use [Accessuse], mientras que en la segunda parte daremos una definición más específica de las sugerencias que nos da WAI-ARIA para este tipo de elementos [WAIARIA].

Non-modal dialogs: tips generales

Estas sugerencias cubren a los elementos non-modal dialog que se abren automáticamente cuando carga la página web (ej. Email registration, Notificaciones de cookies, etc.) o cuando el usuario se mantuvo un cierto tiempo en la página; normalmente, invitaciones a encuestas, diálogos de chat con un representante, u ofertas promocionales. Las sugerencias son:

- **Consideración de manejo del foco:** Según el contenido y el contexto de uso, considere si desea o no establecer el foco en el non-modal dialog cuando se abre. En la mayoría de los casos, los non-modal dialogs que no obstruyen otro contenido no deberían capturar el enfoque actual de los usuarios.
- **Hacer que los non-modal dialogs que cubren contenido de la página sean de fácil ocultamiento:** si el non-modal dialog cubre el contenido de la página, es importante que pueda obtener y perder el foco también a través del teclado (el foco pasa a otro elemento de la página). Entonces es mejor establecer desde el principio el foco en el non-modal dialog para que los usuarios del teclado puedan descartarlo fácilmente. Una excepción son las notificaciones en una franja en la parte inferior de la página; aquí, los usuarios pueden desplazarse por el contenido para verlo.
- **Se debe anunciar la aparición de un non-modal dialog auto generado por la página:** Los diálogos que aparecen automáticamente luego de ciento tiempo deben ser anunciados con el uso del atributo *aria-live=”polite”* así los usuarios de screen reader son notificados de la aparición de estos elementos sin que se pierda el foco actual.

Patrón de diseño WAI-ARIA para los non-modal dialogs

ARIA menciona dos pasos principales para desarrollar estos elementos, que son:

1. Asignación de roles.
2. Asignación de propiedades y relaciones importantes.

Asignación de roles

El elemento dialogo, en general de tipo , debe contener el rol *dialog*, el cual ayuda a las tecnologías de asistencia a identificar el contenido del dialogo como *agrupado y separado del resto del contenido de la página*. Aun así, agregando *role=”dialog”* solamente no es suficiente para hacer que un elemento dialogo sea accesible. Adicionalmente, se deben realizar las siguientes acciones:

- El dialogo debe estar debidamente etiquetado.
- El foco del teclado debe estar manejado correctamente.

Asignación de propiedades, etiquetas y relaciones

Etiquetar a los elementos dentro del diálogo proporcionará información contextual sobre los elementos interactivos dentro del diálogo.

Si un diálogo ya tiene un elemento de título visible, el texto dentro de ese elemento se puede usar para etiquetar el diálogo. La mejor manera de lograr esto es usando el atributo *aria-labelledby* en el elemento *role="dialog"*. Además, si el diálogo contiene texto descriptivo adicional además del título del diálogo, este texto se puede asociar con el diálogo mediante el atributo *aria-describedby*.

Este enfoque se muestra en el fragmento de código a continuación, Figura 6.4.1:

```
<div role="dialog" aria-labelledby="dialog1Title" aria-describedby="dialog1Desc">
  <h2 id="dialog1Title">Your personal details were successfully updated</h2>
  <p id="dialog1Desc">You can change your details at any time in the user account section.</p>
  <button>Close</button>
</div>
```

Figura 6.4.1: Captura de código HTML para ejemplificar el patrón de diseño propuesto por WAI-ARIA para los non-modal dialog.

La combinación entre el rol *dialog* de ARIA y las técnicas de etiquetado logran que el screen reader anuncie la información del dialogo cuando el foco es dirigido a él.

Manejo del foco

ARIA sugiere que un dialogo deber poseer al menos un elemento interactivo para controlar el foco, por lo general hay un botón “Cerrar”, “Enviar” o “Cancelar”. Los cuales sirven para volver al contenido principal. Para los *non-modal dialog* el foco no debería ser capturado dentro del mismo, el usuario debe poder continuar haciendo foco en otros elementos del sitio web mientras el o los *non-modal dialog* siguen visibles. Distinto es el caso de los elementos modales, en los cuales el foco es (intencionalmente) capturado dentro del elemento y el usuario no puede interactuar con el contenido de la página web hasta no cerrar el modal.

Además, es recomendable proveer a los usuarios un acceso rápido por teclado a todos los diálogos abiertos.

6.5 Herramienta para la detección y reporte de elementos flotantes inaccesibles

6.5.1 Introducción a la herramienta

En este capítulo, nos centramos en describir los tipos de elementos flotantes que puede tener un sitio web y los posibles *bad accessibility smells* que estos elementos pueden generar, complicando la experiencia de usuario para aquellos que utilizan screen readers.

Al igual que para la detección de problemas de accesibilidad en mensajes dinámicos de formularios (Capitulo 5), en la herramienta que se desarrolló en esta tesina, también se incorporó la detección de elementos web flotantes con problemas de accesibilidad.

El flujo de trabajo de esta implementación sigue siendo la misma descripta en la Figura 3.1.1.

La extensión web es la encargada de detectar los elementos inaccesibles, los reporta a la API REST para almacenar la información en la base de datos que luego va a ser consumida por la aplicación de reportes.

Para la implementación de la búsqueda de estos elementos en la extensión web, también se utilizó como punto de partida algunos de los pasos definidos en la herramienta implementada por William Watanabe [Watanabe16]. Los cuales son la detección de los elementos que nos interesan analizar, realizando la búsqueda en el DOM del sitio web. Y luego analizamos los atributos del elemento en busca de propiedades y roles ARIA.

Las siguientes dos secciones describirán el método de detección. En la primera (sección 6.5.2) describiremos la detección al momento en que se ingresa al sitio web. En esta etapa aparecen los elementos non-modal dialog que nos interesan analizar, pero hay algunos que se generan luego de un tiempo de estar en el sitio web, o luego de que se dispara algún evento determinado por el usuario. Para ese último caso se buscó una forma para implementar una búsqueda dinámica, la cual describiremos en la segunda sección (sección 6.5.3).

6.5.2 Descripción del método de búsqueda al ingresar a un sitio web

Los elementos que se deseaban detectar en esta búsqueda de *non-modal dialog* eran los que aparecen en pantalla ni bien carga la página web. Estos elementos no se muestran por algún evento de usuario sino de forma automática por la misma página, todos los ejemplos citados en la sección 6.3 son desplegados de forma automática. Sabiendo esto y observando que estos sitios web de ejemplo demoraban entre 2 a 3 segundos en renderizar todos sus componentes incluyendo a los elementos flotantes, en la implementación se definió el inicio de la búsqueda de elementos flotantes luego de que termine de cargarse la página web por completo.

Para lograr esto, se utilizó un *Event Listener* de JavaScript para escuchar el evento **load**, el cual es disparado cuando la página completa ha sido cargada, incluyendo todos sus recursos como los estilos e imágenes.

```
// realizo la primera busqueda cuando termina de cargar completamente la pagina
function onLoadCompleteDetection(reportedElements) {
  window.addEventListener('load', (event) => {
    const elemsInicio = obtenerElemAlInicio();
    const elemsInicioFinal = eliminarElementosOcultos(elemsInicio);
    elemsInicioFinal.forEach(elem => isFloatDivAccesible(elem, reportedElements));
  });
}
```

Figura 6.5.1: Fragmento del código JS de la extensión web, en cual mostramos la definición del método *onLoadCompleteDetection()*.

Como se observa en la Figura 6.5.1, utilizamos el evento load para saber en qué momento se puede ejecutar la búsqueda, ya que todos los elementos de la página web deberían estar presentes en la estructura del DOM y completamente cargados.

Luego de ese momento, empezamos a analizar el DOM en busca de elementos web flotantes. Durante la investigación para este caso, encontramos que todos los elementos analizados manualmente en los ejemplos de la sección 6.3, están definidos con etiquetas ** **. Por ende, tenemos un punto de partida en nuestra búsqueda y empezar a filtrar por tipo de elemento.

Sabiendo que los elementos flotantes que buscamos son <*div*>, debemos encontrar un patrón de diseño que los caracterice y diferencie de los demás *DIVs* de la página. Los modales generalmente poseen estilos CSS particulares que hacen que aparezcan por encima del contenido, diferenciándose del contenido principal. A continuación, describiremos cuáles son los estilos CSS que tienen en común todos los casos analizados.

Estilos CSS tienen en común los pop-ups / modales

- **z-index**: mayor a 0 para posicionarse por encima del contenido principal de la página.
- **Right**, **Bottom**, **Left**, **Top**: para posición al elemento en alguna esquina de la pantalla, por ejemplo. Siempre están fijos en una misma posición de la pantalla mientras el contenido principal de la página se mueve con la navegación en el contenido.
- **Position:** fixed.
- **Display:** flex.

Entonces, como se observa en la Figura 6.5.1, en el método *obtenerElemAlInicio()* se realiza el filtrado de elementos DIVs. En la misma búsqueda también nos quedamos con los que tengan definidos los estilos CSS mencionados anteriormente, que pueden obtenerse a través del método **getComputedStyle(elem).getPropertyValue()** y así comprobar si cumplen con el patrón de estilos que definimos. Esta comprobación se realiza con la función JS de la Figura 6.5.2.

```
function esElementoFlotante(zIndex, position) {
    return (zIndex > 1) && (position === 'fixed');
}
```

Figura 6.5.2: Código JS de la función que realiza un chequeo simple el valor de los atributos.

En principio, en la búsqueda nos quedamos con los elementos que posean los atributos con los siguientes valores:

- z-index > 1
- position = ‘fixed’

El chequeo de estos valores fue suficiente para detectar con suficiente certeza los elementos que nos interesan. En la sección de validación de la herramienta, se demostrará la eficacia de esta búsqueda.

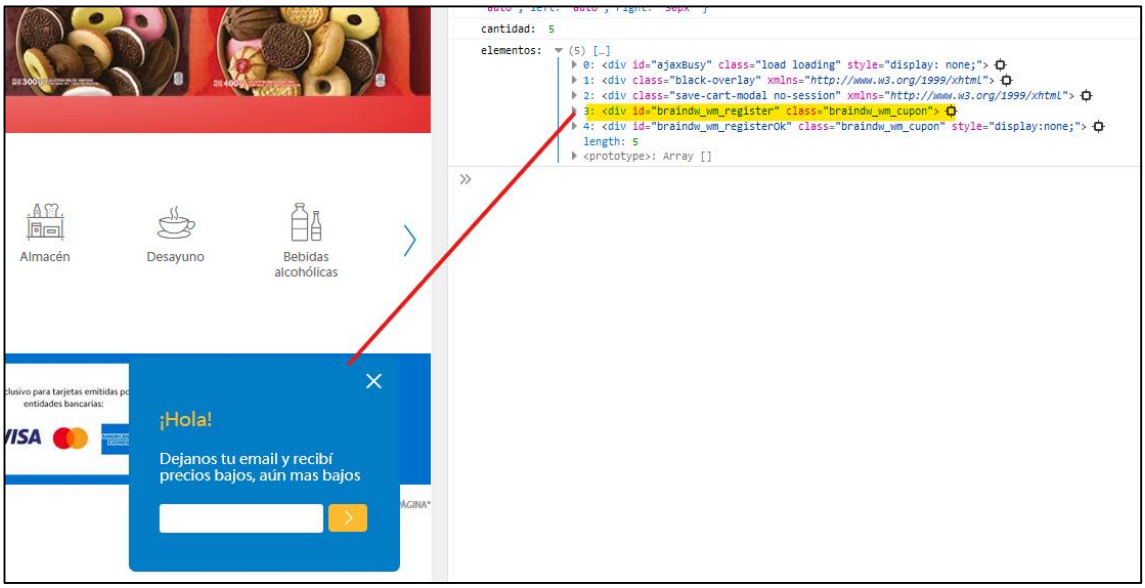


Figura 6.5.3: Ejemplo del resultado de búsqueda con elementos ocultos en el sitio web de Walmart Online. Solamente el elemento #3 es un non-modal dialog, los demás son elementos ocultos.

No obstante, durante el trascurso del testeo de esta búsqueda, nos encontramos con el problema de que en el resultado de la búsqueda habían DIVs flotantes que estaban ocultos, o sea que no eran de nuestro interés reportarlos ya que pueden no ser interactivos para el usuario (como se puede ver en la Figura 6.5.3), y por lo tanto debían ser descartados durante la búsqueda.

Revisando el código HTML de estos elementos no deseados, se encontró que también cumplen un patrón de estilos CSS. Utilizan los siguientes atributos para estar ocultos:

- **visibility**: hidden;
- **display**: none;

Por ende, descartamos de nuestra búsqueda a todos los elementos que posean alguno de esos atributos, ya que con cualquiera de ambos van a ser invisibles para el usuario y *screen readers*. En la Figura 6.5.1, este paso se realiza en el método *eliminarElementosOcultos()*.

Es posible que alguno de los elementos ocultos sean también modales que deban ser analizados, pero en esta búsqueda inicial, nos centraremos en los que están a disposición del usuario al iniciar la página web. En la sección 6.5.3, definiremos la búsqueda dinámica donde incluiremos a estos elementos flotantes de aparición dinámica.

Continuando con la búsqueda de elementos flotantes que se muestran al iniciar la página web, el último paso en la detección, consiste en revisar si los elementos poseen los atributos correspondientes de Rol y ARIA para poder ser accesibles, de esta tarea se encarga el método *isFloatDivAccesible()*, como se ve en la Figura 6.5.1.

Para lograr que la detección sea más efectiva, además de ejecutar una búsqueda ni bien se genera el evento **load**, también realizamos una nueva búsqueda luego de 5 segundos de ingresar a la página. De esta forma nos aseguramos que también sean incluidos en la búsqueda los elementos que aún no están completamente generados en el DOM al terminar de cargar el sitio web. Entonces, al generarse dos búsquedas, debimos asegurarnos de no reportar un mismo elemento inaccesible dos veces; esto se logró mediante el uso de un arreglo donde se van cargando los *ID* (Identificador del elemento HTML) de los elementos que van siendo reportados.

6.5.3 Descripción del método de búsqueda para elementos que se generan dinámicamente

Con la solución descripta en la sección 6.5.3, se lograron capturar y analizar la mayor parte de los elementos flotantes de nuestro interés. Durante la investigación en distintos sitios webs, hemos encontrado que generalmente hay entre 1 y 2 elementos flotantes (*non-modal dialogs*).

Sin embargo, existen algunos casos donde hay *non-modal dialogs* que se muestran dinámicamente al pasar cierto periodo de tiempo dentro de la página web. Cabe destacar que estos elementos no son renderizados por causa de algún evento disparado por parte del usuario, sino que es el mismo sitio web el que automáticamente los muestra.

Una solución que surgió en un principio, fue la de ejecutar una búsqueda cada 10 segundos. Sin embargo, era una solución muy ineficiente, ya que son pocos los casos donde se disparan elementos flotantes automáticamente y en la mayoría de los casos las búsquedas iban a ser sin sentido; además de que el fin de esta tesina no es el de desarrollar soluciones estáticas. Así que se debía encontrar algún tipo de solución dinámica para lograr detectar estos elementos.

Se logró desarrollar e integrar a la búsqueda de elementos flotantes (sección 6.5.3) una solución dinámica basada en la arquitectura definida por Watanabe [Watanabe16], que describimos a continuación.

Utilización del approach de Watanabe para la detección dinámica de elementos flotantes

Para lograr ampliar nuestra detección a elementos flotantes que se despliegan en la página dinámicamente, se utilizó como guía de desarrollo el diseño de la herramienta de detección desarrollada por Watanabe. Como se puede observar en la Figura 2.3.3, la herramienta captura los eventos que se generan en el DOM, se queda con los que son generados en elementos que son de interés para la detección y se analizan sus atributos. En el caso de esta herramienta, su interés era el de detectar Widgets dinámicos los cuales son desplegados por alguna acción del usuario. Para esto, la herramienta analiza los eventos que muestran y ocultan elementos de la página web. Este patrón de búsqueda es el que necesitamos para lograr detectar los non-modal dialogs en nuestra herramienta.

En un principio la solución desarrollada era algo compleja, ya que se decidió observar a los cambios de estilos en elementos DIV, específicamente en los estilos CSS denominados **visibility** y **display**. Estos atributos sirven para ocultar cualquier clase de elemento HTML y son utilizados en elementos que aparecen dinámicamente en la página, los modales son un claro ejemplo de esto. Los valores que pueden tener estos atributos son los siguientes:

- **display**
    - none: se oculta el elemento
    - block, inline, grid, flex, etc… En cualquiera de estos casos el elemento es visible.
- **visibility**
    - visible: El elemento se muestra normalmente.
    - hidden: esconde un elemento, pero deja (vacío) el espacio donde debería aparecer.

Si hubo un cambio del atributo *display* en un elemento determinado, este iba a ser almacenado y si volvíamos a obtener un nuevo cambio de *display* en el mismo elemento, podemos asegurar de que estamos observando a un elemento que se oculta y se muestra por cause de eventos de usuario. La principal desventaja de este método es que un elemento solo va a ser reportado si el usuario lo abre y lo cierra o viceversa. Y esto puede ser que nunca se cumpla, ya que, si uno de estos elementos tiene problemas de accesibilidad, puede que el usuario nunca logre cerrarlos o abrirlos.

Al observar detalladamente la definición de la herramienta de Watanabe [Watanabe16], podemos observar que realiza una búsqueda previa de los elementos que están ocultos, para luego observar si éstos son mostrados al usuario.

En concreto, la solución consistiría en observar los elementos que contienen los atributos:

- **display**: none
- **visibility**: hidden

Por lo tanto, se utilizó la estrategia en nuestra búsqueda, como observamos en la Figura 6.5.4, la función *obtenerElementosOcultos()* retorna los elementos DIV que poseen alguno de los atributos de ocultamiento listados anteriormente.

```
function obtenerElementosOcultos() {
    let divs = getBodyDiv();
    return [...divs].filter(elem => {
        // Attributos de ocultamiento
        let display = window.getComputedStyle(elem).getPropertyValue('display');
        let visibility = window.getComputedStyle(elem).getPropertyValue('visibility');
        return visibility === 'hidden' || display === 'none';
    });
}
```

Figura 6.5.4: Código de la función JS con la cual se obtienen los elementos ocultos.

Al tener detectados los elementos a los cuales vamos a observar, al igual que en la herramienta de Watanabe, se fijan los Mutation Observer a estos elementos ocultos para lograr capturar si en algún momento pasan a ser visibles.

Esta detección funciona dinámicamente, por ende, va a detectar aquellos elementos flotantes que cambien de estado oculto a desplegado en cualquier momento.

Cuando se detecta que un elemento estaba oculto y luego pasa a ser visible, se analiza si es un elemento de tipo flotante. Para eso se analizan sus atributos CSS, tal cual muestra la figura 6.5.2. Y si cumple, se busca entre sus atributos la ausencia de roles y atributos *ARIA*. Si no existe ninguno de estos atributos, se reporta el elemento a la *API REST* para que luego sea mostrado en el reporte.

Descubrimiento inesperado

Esta nueva solución dinámica apuntaba a lograr detectar los non-modal dialogs que son disparado automáticamente por la misma página web. Aunque durante las pruebas, vimos que también detectaba algunos tipos de modales y elementos flotantes que no entraban en esta búsqueda. Estos elementos son aquellos que pasan de estar ocultos y a mostrarse debido a alguna acción disparadora que realiza el usuario. En la siguiente sección mostraremos varios ejemplos de esto.

6.6 Resultado final de la herramienta para la detección de elementos flotantes inaccesibles

Para mostrar el resultado final de la herramienta para detectar elementos flotantes inaccesibles, se ejecutó la misma en varios sitios web, con los siguientes resultados.

6.6.1 Sitio web con elementos flotantes #1

Página de inicio de Walmart Online

URL

https://www.walmart.com.ar/

Comportamiento de NVDA

Mientras navegamos por la página con NVDA, logramos entrar al elemento, pero solamente detectamos un input y un botón *submit*. Esto termina siendo muy confuso ya que aparecen durante la navegación sin darle ningún tipo de contexto al usuario. Esto se debe a que no hay ningún atributo para notificar e informar al usuario sobre el contenido y utilidad de este elemento.

Elementos Flotantes

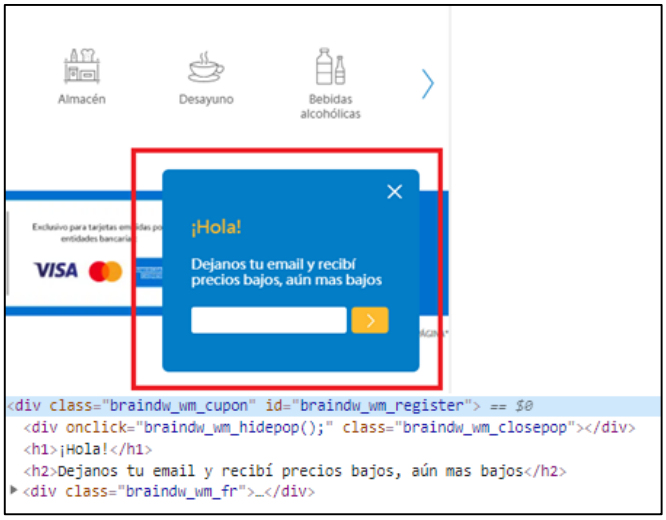


Figura 6.6.1: Único non-modal dialog desplegado al iniciar la página web, junto a su código HTML.

Resultado final de la herramienta

Investigamos en detalle el código HTML del elemento de la Figura 6.6.1, y no posee atributos ROL y ARIA para hacerlo accesible.

La herramienta detecta al elemento, lo analiza y lo clasifica como elemento no accesible.

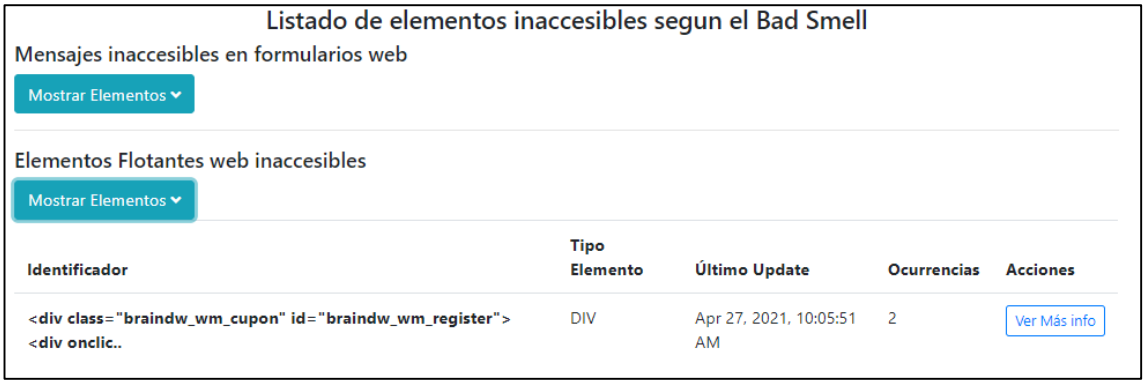


Figura 6.6.2: Resultado para la detección en el Ejemplo #1

En la figura 6.6.3, podemos observar un elemento flotante que es desplegado por una acción por parte del usuario. La herramienta no logró detectar este elemento, en principio la razón era porque está declarado como un elemento *<section>*, y la búsqueda que realiza la herramienta es sobre elementos * * y *<nav>*, pero ampliamos la búsqueda para incluir elementos *<section>*, aun así, vimos que, por algún motivo desconocido, los cambios de atributos de este modal no eran capturados por el *Mutation Observer* asignado al elemento. Queda para trabajo futuro, tratar de lograr capturar esos eventos de cambio de CSS en este elemento.

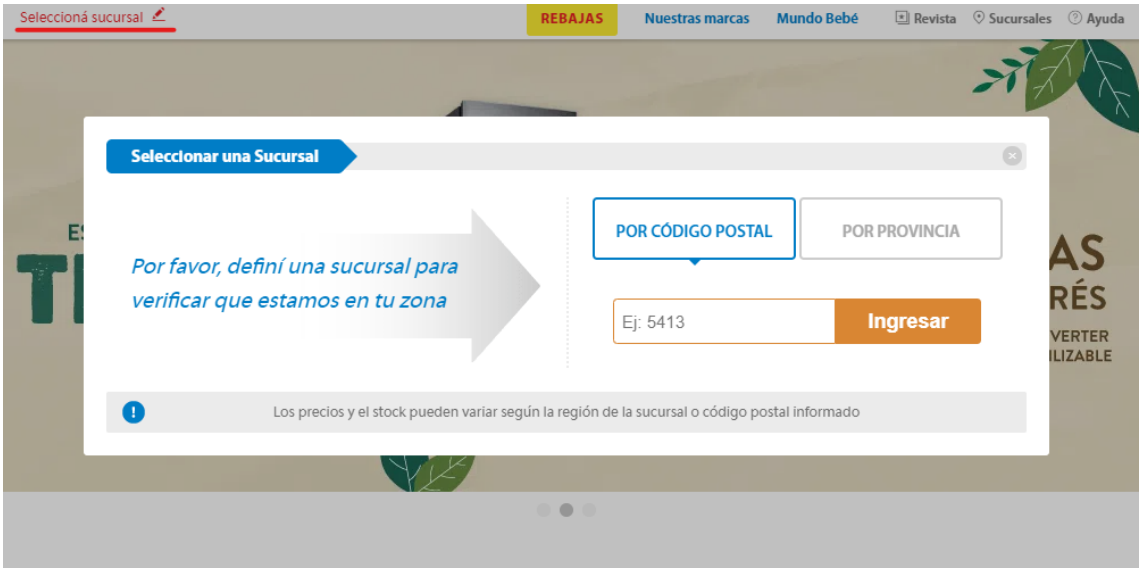


Figura 6.6.3: Elementos flotante desplegado al entrar al link “Seleccioná sucursal”

6.6.2 Sitio web con elementos flotantes #2

Página de inicio de ARBA Online

URL

https://www.arba.gov.ar/

Comportamiento de NVDA

En la página detectamos visualmente un elemento flotante, el asistente virtual (Figura 6.6.4). Para llegar a él, debemos llegar al final de la página para lograr interactuar con el elemento. Cuando logramos posicionarnos sobre el mismo, nos lee una descripción algo confusa. Al presionar el botón *enter* sobre el mismo, nos despliega un nuevo panel flotante con bastante funcionalidad. NVDA no nos avisa que se desplegó el nuevo elemento.

Lo mismo sucede con el menú contraído (Figura 6.6.5), el cual se muestra al presionar la tecla *enter* sobre el elemento. Se despliega un menú, pero no se da ningún tipo de aviso al usuario.

Elementos Flotantes

Figura 6.6.4: Asistente virtual de Arba, oculto y desplegado.

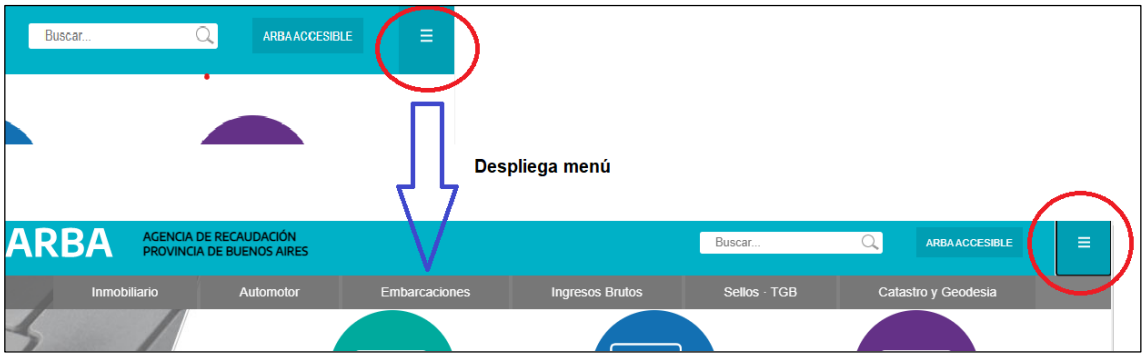


Figura 6.6.5: Menú principal, oculto y desplegado.

Resultado final de la herramienta

Investigamos en detalle el código HTML de los elementos de las figuras 6.6.4 y 6.6.5, y no poseen atributos ROL y ARIA para hacerlos accesibles como podemos observar en la Figura 6.6.6.

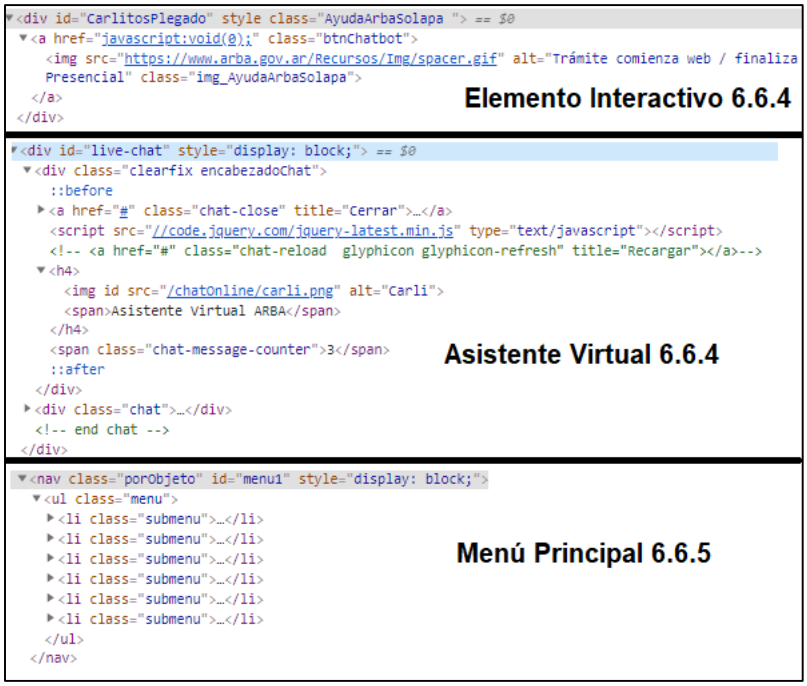


Figura 6.6.6: Código HTML de los elementos de las Figuras 6.6.4 y 6.6.5.

La herramienta logra detectar a los elementos, luego analiza sus atributos, y al no poseer ningún atributo ARIA o Rol, los clasifica como elementos no accesibles, figura 6.6.7.

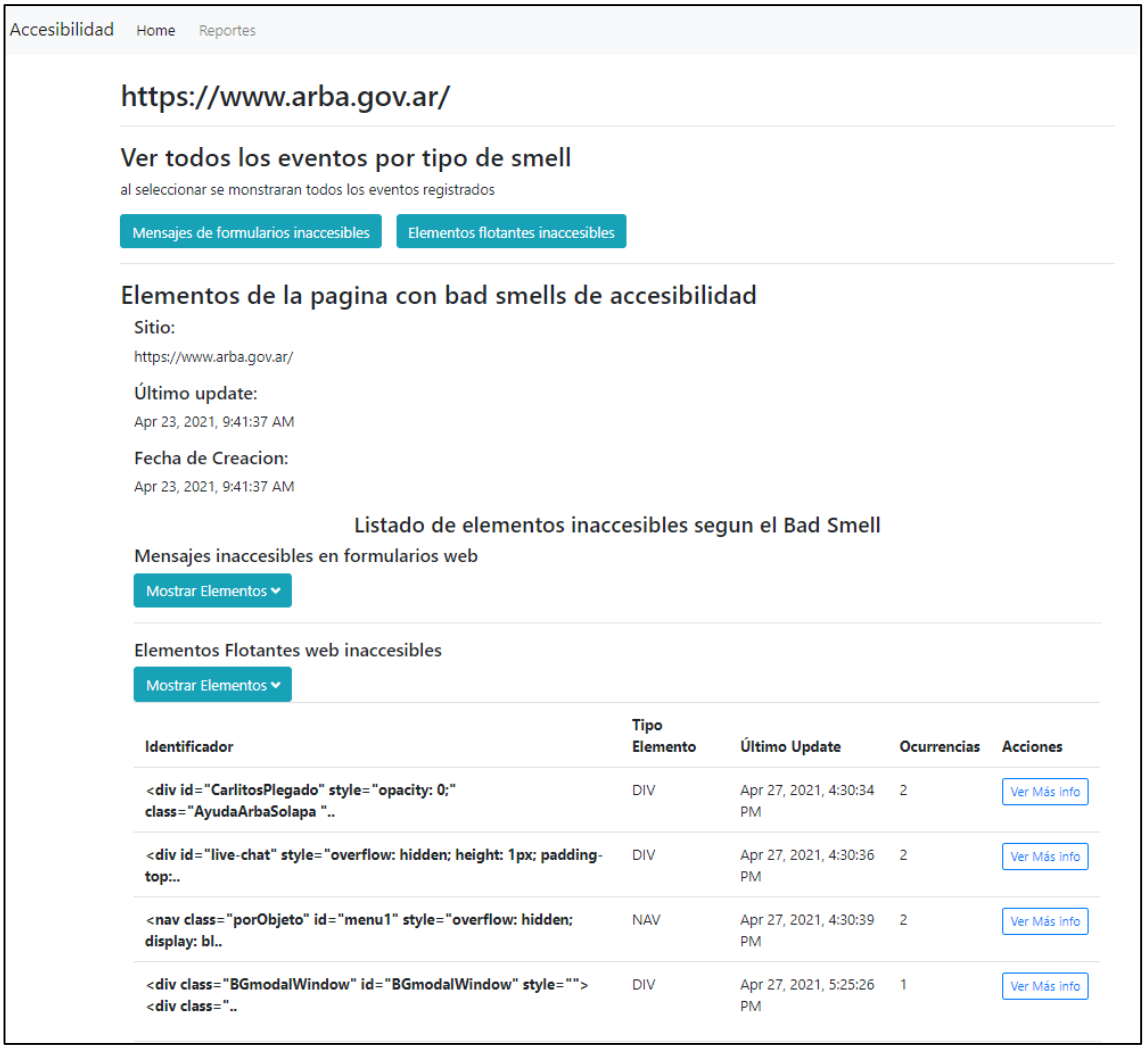


Figura 6.6.7: Reporte generado por la herramienta de detección.

Como podemos observar en la Figura 6.6.7, además de tener presentes en el reporte: el menú, el elemento para desplegar el asistente y el asistente virtual desplegado, vemos que hay un con el ID, *BGmodalWindow*. Este es un modal que se despliega al presionar el botón “Enviar Por Mail”, Figura 6.6.8. Es un elemento que carece de roles y atributos ARIA.

En un principio, creíamos que en el algoritmo de detección no íbamos a contemplar a los modales que se generen dinámicamente debido a su complejidad. Sin embargo, al finalizar el desarrollo del algoritmo, tuvo como resultado el poder lograr detectar no solo los *non-modal dialogs*, sino también modales y algunos menús desplegables. Esto fue algo muy positivo ya que en la búsqueda podemos abarcar más elementos con un mismo algoritmo de detección.

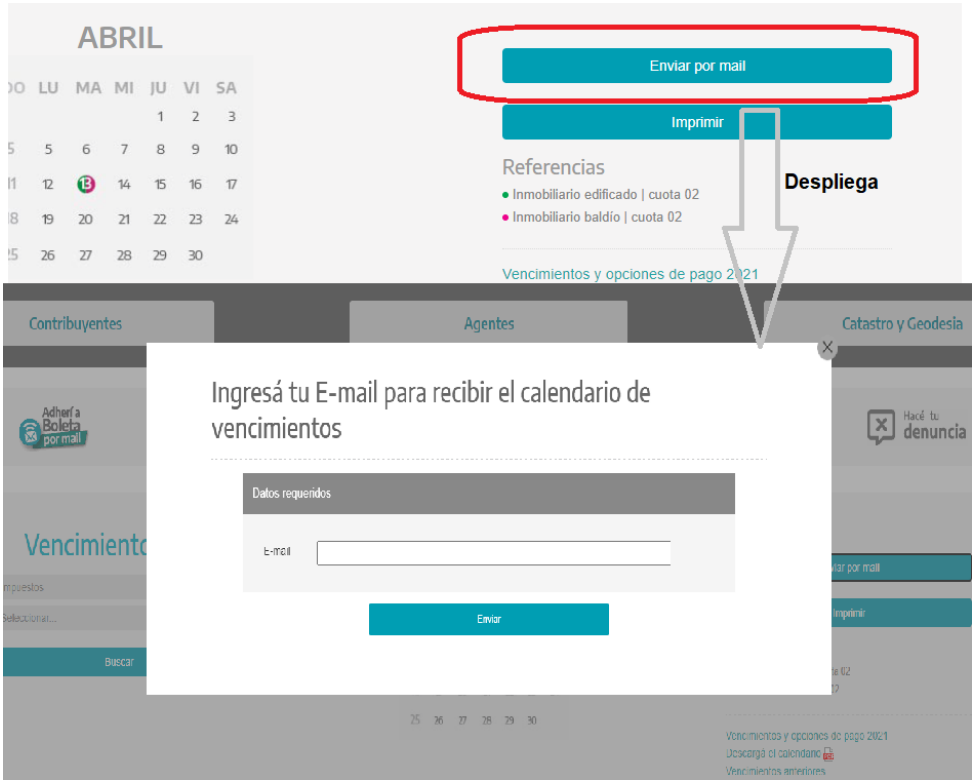


Figura 6.6.8: Modal desplegado al presionar tecla ENTER sobre el botón “Enviar por mail”.

6.6.3 Sitio web con elementos flotantes #3

Página de inicio de APR Online

URL

http://www.apronline.gov.ar/

Comportamiento de NVDA

Ni bien entramos a la página web, nos aparece un modal desplegado, como podemos ver en la figura 6.6.9. Este modal cubre el contenido de la página con una capa oscura, esto quiere decir que el contenido de la página no es accesible mientras el modal este desplegado. Sin embargo, esto no es así, mientras el modal está en el medio de la página, el usuario puede seguir interactuando con los demás elementos de la página. Más allá se ello, cuando se despliega un modal el usuario debe ser notificado sobre esto, y en este caso no sucede. Hay un modal en mitad de la pantalla, pero el usuario no es advertido de esto.

Caso contrario sucede con el modal que se muestra al interactuar con el botón “Escribinos”, Figura 6.6.10, ni bien lo abrimos el foco se posiciona en el modal, nos lee su título y solo podemos interactuar dentro del mismo modal. Es un resultado esperado cuando se aplican los atributos correctos según ARIA, más abajo en esta misma sección, detallaremos el resultado de la herramienta de detección y daremos más detalles sobre este modal.

También tenemos otros tres elementos flotantes en la página, como vemos en la Figura 6.6.11. Si bien hay que llegar al fondo de la página web para acceder a ellos, NVDA los menciona al usuario y se puede interactuar con ellos.

Elementos Flotantes

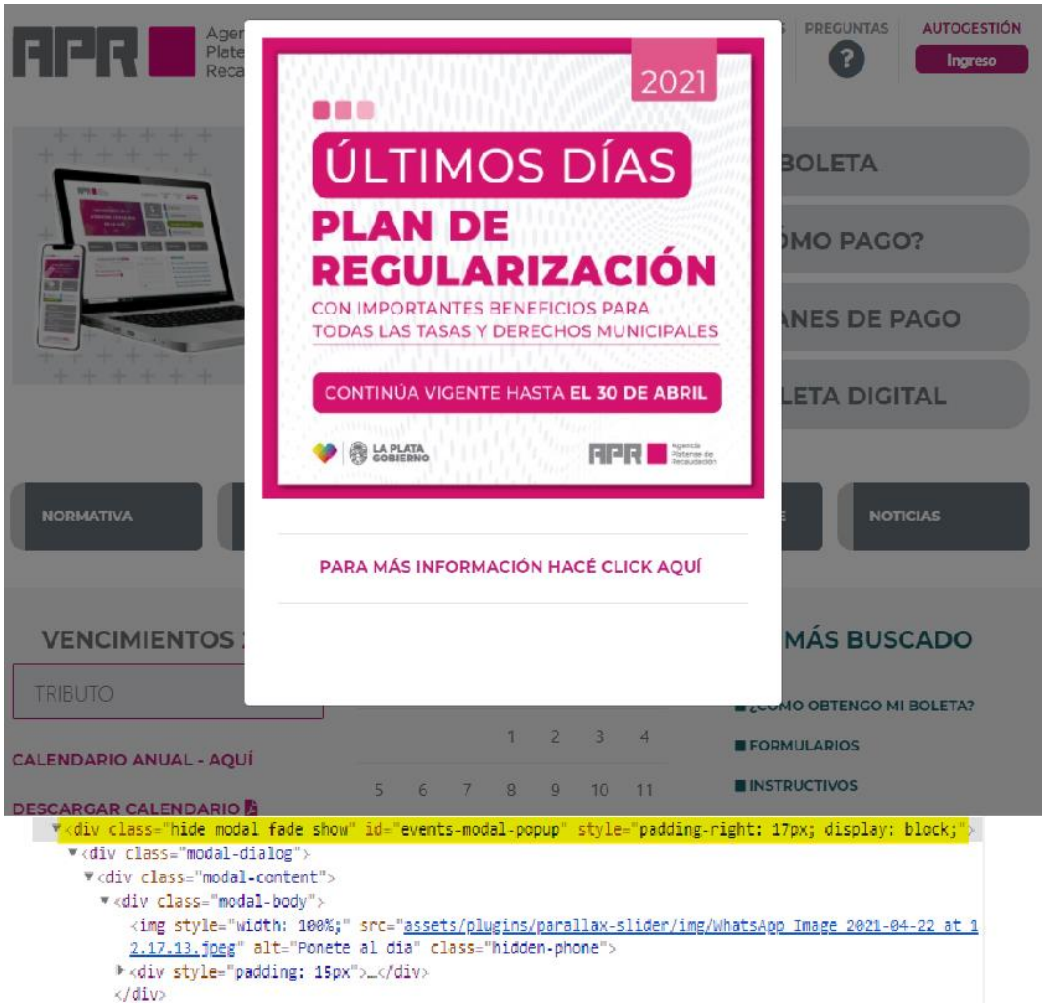


Figura 6.6.9: Modal al inicio de la página web de APR Online.

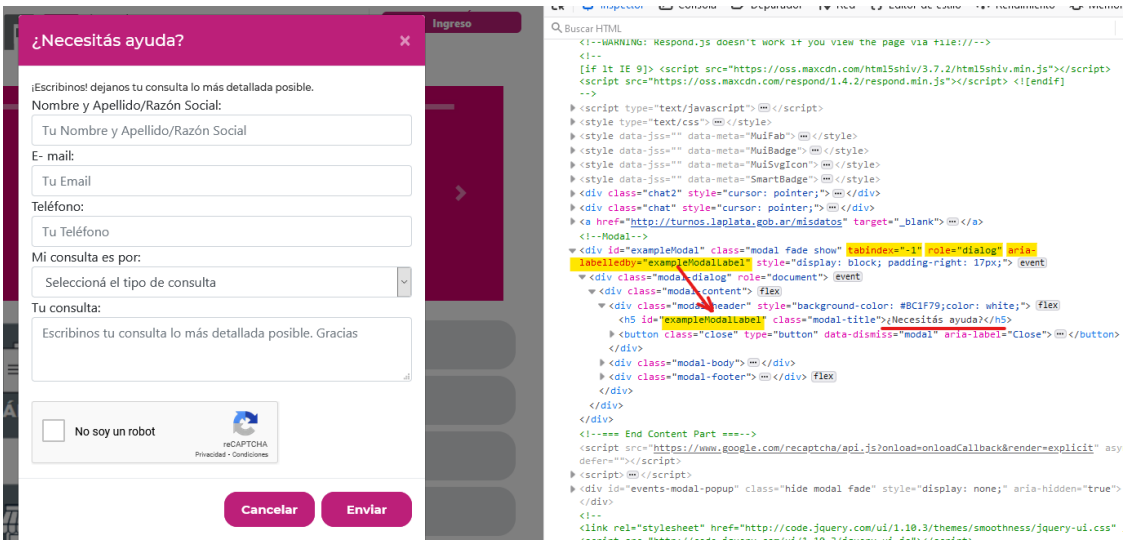


Figura 6.6.10: Modal que desplegado al presionar tecla ENTER en botón “Escribinos”.

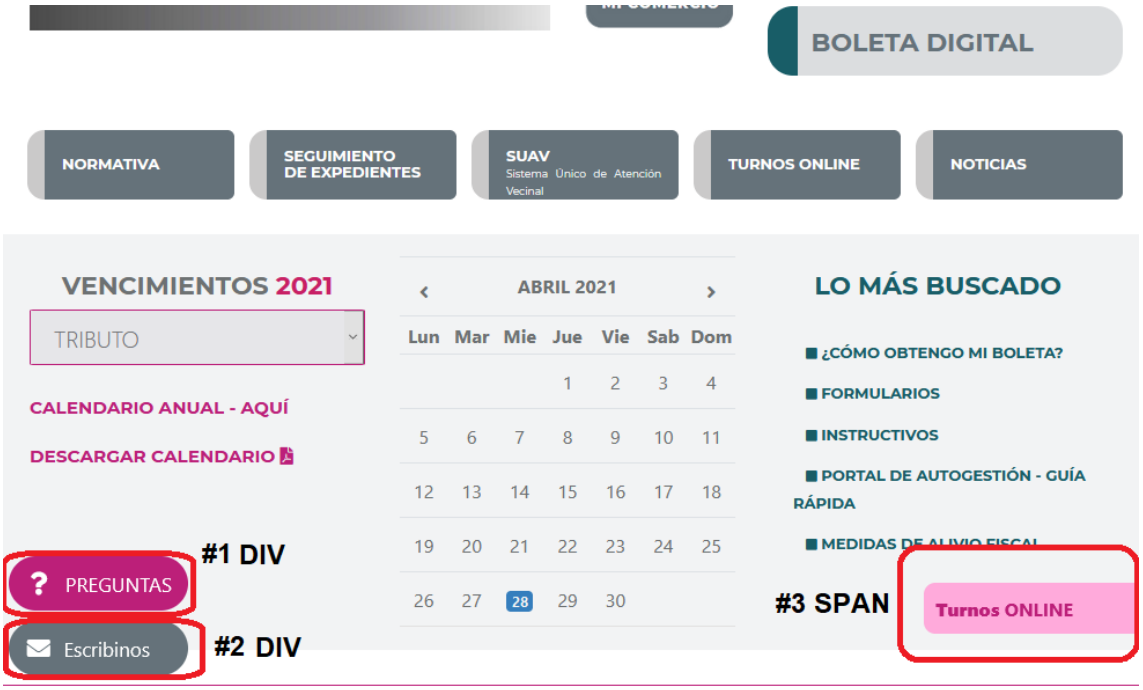


Figura 6.6.11: Elementos *non-modal dialogs* presentes desde el inicio del sitio web.

Resultado final de la herramienta

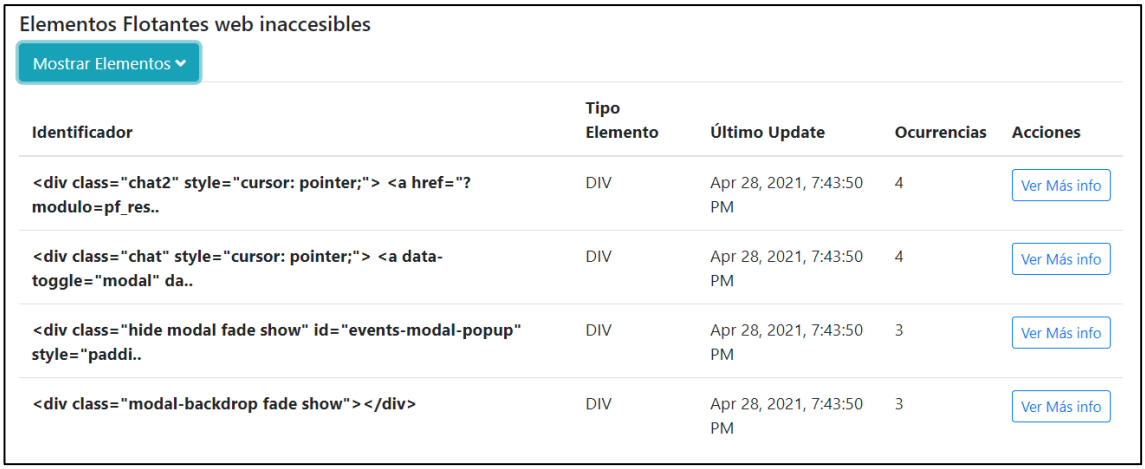


Figura 6.6.12: Resultado final de la detección de elementos flotantes con problemas de accesibilidad.

Como podemos observar en la Figura 6.6.12, vemos que los primeros dos resultados:

1. <div class="chat2"…
2. <div class="chat" …

Referencian a los elementos flotantes #1 y #2 de la Figura 6.6.11. Esto quiere decir que la herramienta los logró detectar como elementos flotantes y los clasificó como inaccesibles. No obstante, este resultado en parte es erróneo ya que estos elementos declarados como , no poseen ningún rol o atributo ARIA para darle un mejor contexto al usuario. Sin embargo, terminan siendo accesibles ya que en su contenido poseen un elemento *<button>* el cual SI posee el atributo *aria-label*, el cual es leído por NVDA.

Como podemos observar en la figura 6.6.13, el elemento botón está dentro de varios elementos los cuales carecen de atributos ARIA para lograr mejorar la accesibilidad del elemento, así que el reporte del elemento es de ayuda, pero en un trabajo futuro se puede mejorar la herramienta de detección para no reportar elementos que tengan hijos con atributos apropiados de ARIA. Realizar esta detección extra puede generar una sobrecarga en páginas con varios elementos flotantes ya que como vemos en la Figura 6.6.13, hay que analizar varias capas para llegar al elemento que logra aportar el atributo aria para que el principal sea accesible.

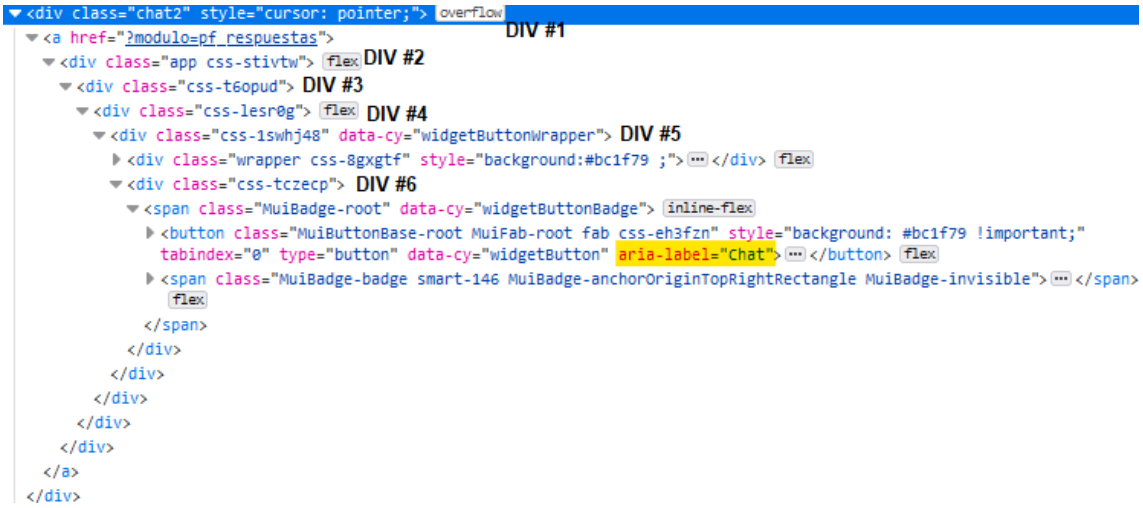


Figura 6.6.13: Captura del código del botón “Preguntas” para demostrar la cantidad de capas que se necesitan procesar para llegar al botón.

Continuando con el análisis del resultado de la herramienta mostrado en la Figura 6.6.12, vemos que los últimos dos elementos reportados:

a) **class="hide modal fade show" id="events-modal-popup"**

b) **class="modal-backdrop fade show"**

Están relacionados con el modal de la Figura 6.6.9, el modal en sí que contiene el contenido es el elemento reportado a), y está correctamente reportado ya que no posee ningún atributo o rol ARIA para ser accesible. El elemento b) también carece de estos atributos, pero no es correcto reportarlo ya que es la sombra del modal. Y fue reportado por la herramienta ya que posee las mismas características de un elemento flotante. Solo se detectó este único caso, pero sería preferible para una futura ampliación de la herramienta, lograr que se ignoren este tipo de elementos durante la detección de elementos flotantes.

Como mencionamos anteriormente, el modal desplegable de la Figura 6.6.10, es accesible ya que como vemos en su código, posee no solo el atributo *rol=dialog*, sino que también el atributo *aria-labelledby* que relaciona el modal con su respectivo título, el cual es leído ni bien el modal es desplegado. Y como es de esperase el elemento no fue reportado como inaccesible.

6.6.4 Sitio web con elementos flotantes #3

Página de inicio de Diario El Día, de La Plata

URL

https://www.eldia.com/

Comportamiento de NVDA

Como se observa en la Figura 6.6.13, vemos que este sitio web contiene una mayor cantidad de elementos *non-modal dialogs* comparado con los ejemplos anteriores. Además de esto, contiene una gran cantidad de contenido relacionado a noticias periodísticas, así que la lectura del sitio e interacción con sus elementos se vuelve un trabajo tedioso, pero en este capítulo solo nos interesa la búsqueda y análisis de un único tipo de elementos, elementos flotantes. Ni bien se ingresa a la página web y esta termina de cargar, se observan hasta 6 diferentes elementos flotantes, numerados en la Figura 6.6.13.

Al navegar sobre la página usando NVDA, el pop-up #1 es interactivo, se puede navegar sobre sus botones, pero no da ningún contexto al usuario sobre cuál es el uso del pop-up.

Sobre los banners de publicidad #3 y #4, el usuario pasa sobre los elementos, pero NVDA no da ninguna descripción para orientar al usuario y comunicar el contenido de ambas publicidades.

El elemento #6 no es accesible, no hubo forma de lograr pararse sobre él utilizando el teclado. Mientras que el elemento #5, también un pop-up sobre publicidad, NVDA logra posicionarse sobre el elemento, pero no menciona nada.

Al desplegar los modales para Iniciar Sesión y Registrarse, Figura 6.6.14, la experiencia de usuario empeoro aún más. NVDA nunca mencionó al usuario que los modales fueron desplegados, y además el foco no se pasa automáticamente a los modales, sigue en el contenido principal de la página y se puede seguir interactuando con el mismo aun estando los modales desplegados. Además, ninguno de los modales puede ser cerrado presionando la tecla escape, solamente pueden ser cerrados al interactuar con el botón para cerrar el modal.

Elementos Flotantes

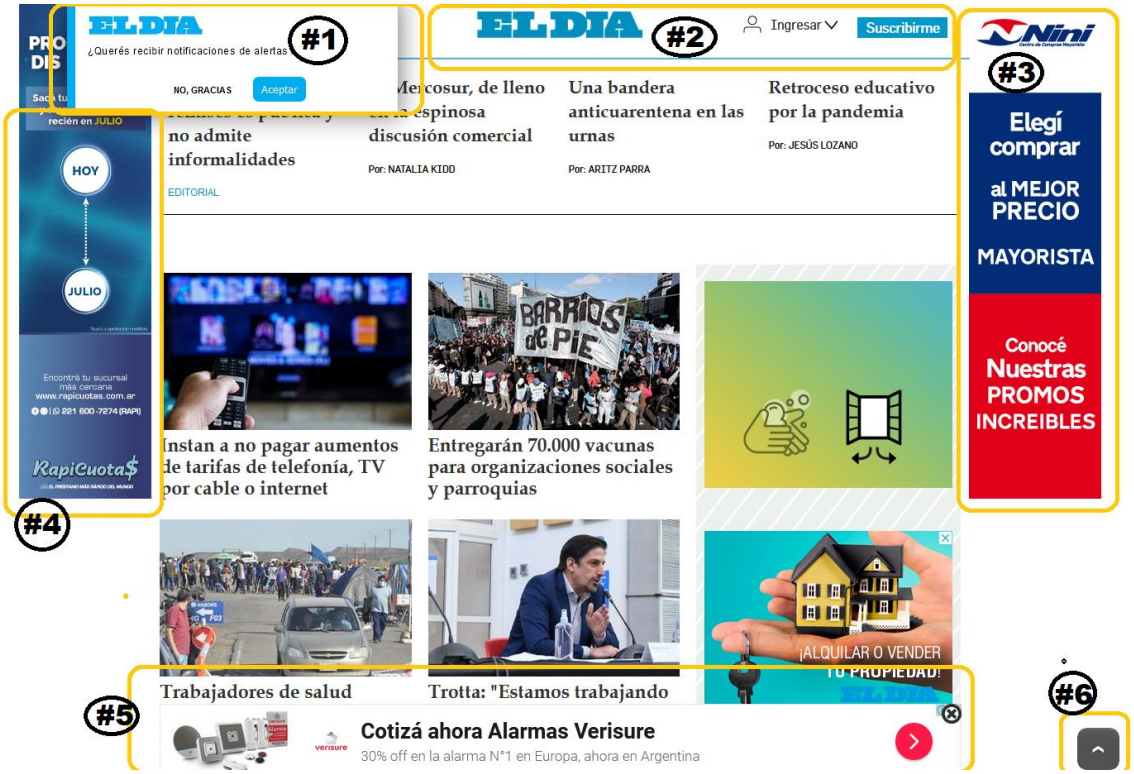


Figura 6.6.13: Captura de la página principal de Diario El Día, donde se numeraron los diferentes *non-modal dialogs* presentes.

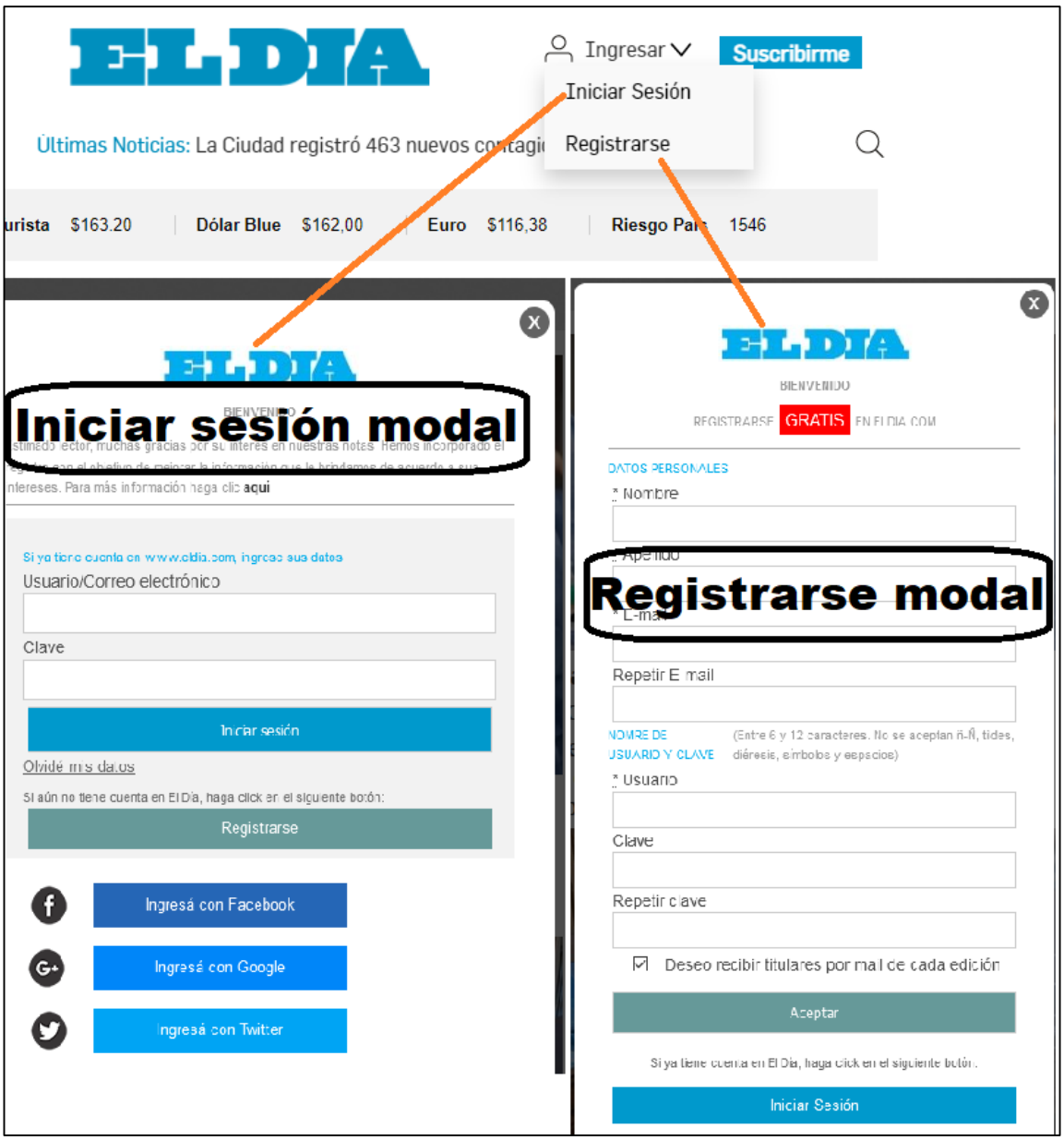
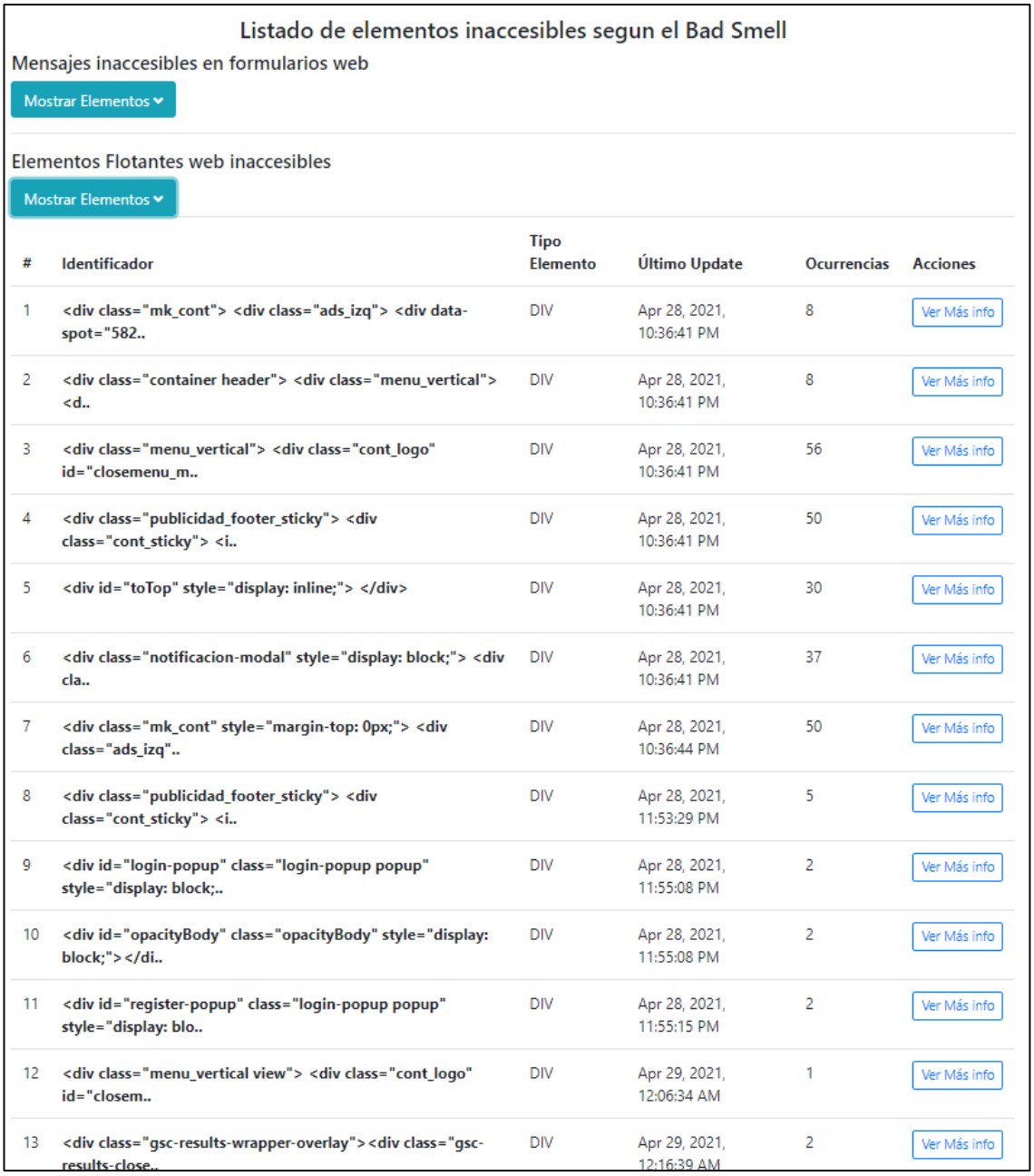


Figura 6.6.14: Captura de la página principal de Diario El Día, donde se enumeraron los diferentes *non-modal dialogs* presentes.

Resultado final de la herramienta

El resultado de la herramienta en este caso detectó una cantidad más grande de elementos flotantes inaccesibles comparado a los casos anteriores, como se puede observar en la Figura 6.6.15. Luego de la siguiente figura, detallaremos este resultado.



Listado de elementos inaccesibles segun el Bad Smell

Mensajes inaccesibles en formularios web

Mostrar Elementos

Elementos Flotantes web inaccesibles

Mostrar Elementos

#	Identificador	Tipo Elemento	Último Update	Ocurrencias	Acciones
1	<div data-spot="582..	DIV	Apr 28, 2021, 10:36:41 PM	8	Ver Más info
2	<d..	DIV	Apr 28, 2021, 10:36:41 PM	8	Ver Más info
3	<div class="cont_logo" id="closemenu_m..	DIV	Apr 28, 2021, 10:36:41 PM	56	Ver Más info
4	<i..	DIV	Apr 28, 2021, 10:36:41 PM	50	Ver Más info
5		DIV	Apr 28, 2021, 10:36:41 PM	30	Ver Más info
6	<div cla..	DIV	Apr 28, 2021, 10:36:41 PM	37	Ver Más info
7	<div class="ads_izq"..	DIV	Apr 28, 2021, 10:36:44 PM	50	Ver Más info
8	<i..	DIV	Apr 28, 2021, 11:53:29 PM	5	Ver Más info
9	<div id="login-popup" class="login-popup popup" style="display: block;..	DIV	Apr 28, 2021, 11:55:08 PM	2	Ver Más info
10	</di..	DIV	Apr 28, 2021, 11:55:08 PM	2	Ver Más info
11	<div id="register-popup" class="login-popup popup" style="display: blo..	DIV	Apr 28, 2021, 11:55:15 PM	2	Ver Más info
12	<div class="cont_logo" id="closem..	DIV	Apr 29, 2021, 12:06:34 AM	1	Ver Más info
13	<div class="gsc-results-close..	DIV	Apr 29, 2021, 12:16:39 AM	2	Ver Más info

Figura 6.6.15: Captura del reporte final generado por la herramienta de detección de elementos flotantes inaccesibles.

En el reporte, vemos que los elementos #1 y #7, son los elementos de publicidad #3 y #4 de la Figura 6.6.13. NVDA lee los atributos ALT de las imágenes que contienen ambos , pero sería de gran ayuda que ambos elementos tengas un ROL y atributos ARIA asignados para mejorar la accesibilidad de ambos elementos. En la Figura 6.6.16 vemos el código HTML de uno de los elementos, y vemos que no posee ningún rol u atributos ARIA, así que el reporte de ambos es correcto.

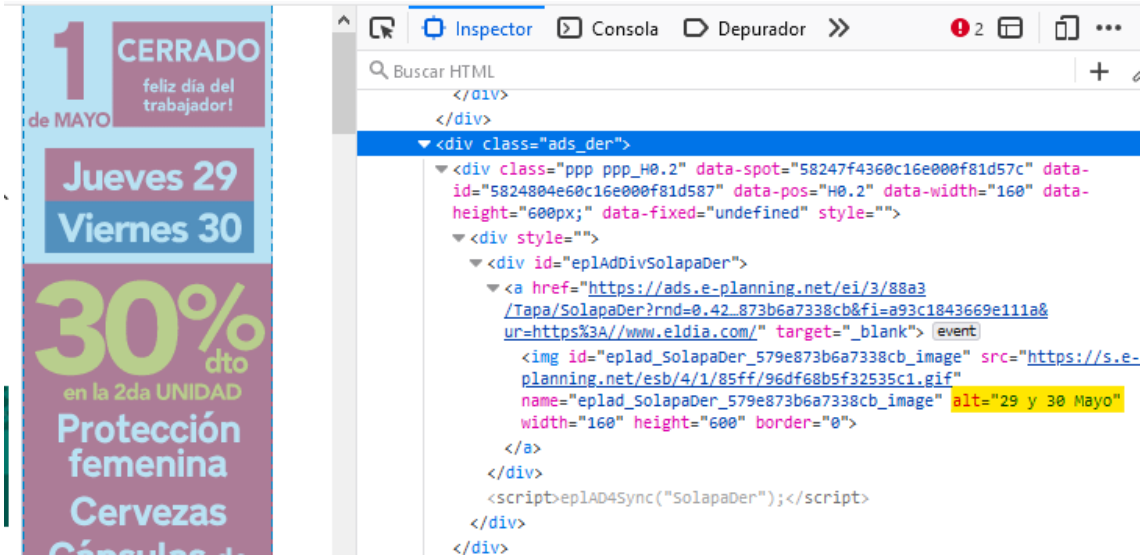


Figura 6.6.16: Captura del código HTML de los elementos flotantes de publicidad.

En la Figura 6.6.13, el elemento flotante #1 para recibir notificaciones del diario, también es reportado correctamente en el reporte. En el reporte es el elemento #6, y al revisar su código HTML vemos que también carece de los atributos necesarios para hacerlo accesible, Figura 6.6.17, y al igual que en los elementos anteriores de publicidad, NVDA solo lee el atributo ALT de la imagen que contiene el logo del diario.

Figura 6.6.17: Captura del código HTML del elemento flotante para aceptar recibir notificaciones.

¿Fueron reportados los modales que se muestran dinámicamente?

Si, tanto el modal de Iniciar Sesión como el de Registrarse fueron reportados y están presentes en el reporte. Elementos #9 y #11. Como vemos en la Figura 6.6.18, en el código HTML de ambos elementos no hay atributos ARIA presentes.

Figura 6.6.18: Captura del código HTML de los elementos modales de Login y Registrarse, que se disparan por acción del usuario dinámicamente.

CAPÍTULO 7

CONCLUSIÓN Y TRABAJOS FUTUROS

7.1 Conclusiones

La accesibilidad web no es algo sencillo de lograr si la misma no está incluida en el proceso de desarrollo. Esto produce que los desarrolladores no tengan en cuenta a la accesibilidad como parte del desarrollo de una página web, por ende, tampoco van a ver la necesidad de capacitarse en el tema.

Esto tiene como consecuencia que se comience a dar importancia a la accesibilidad web luego de que el costo de no incluirla en el desarrollo comience a aumentar. El primer impacto va a ser la disminución de usuarios en el sitio web debido a que no pueden utilizarlo ni acceder a su contenido. Además, existen sanciones que se pueden llegar a recibir por no proveer un producto accesible; en este trabajo se citaron varios casos de empresas sancionadas por falta de accesibilidad en sus productos web.

Para ayudar a los desarrolladores a lograr proveer sitios webs accesibles, actualmente existen varias herramientas online y gratuitas para ejecutar análisis de código en busca de problemas de accesibilidad. Sin embargo, están limitadas a solo poder encontrar aquellos problemas de accesibilidad que pueden ser detectados a través del análisis estático de código.

En esta tesina se buscó brindar a los desarrolladores y dueños de páginas web, una herramienta que facilite la forma en que pueden encontrar diferentes tipos de elementos webs que generen *accessibility smells*. El fin de esta herramienta no es solo la de detectar los elementos problemáticos, sino además almacenar su información para luego generar reportes de accesibilidad que sean de fácil lectura.

También se citaron varios trabajos relacionados como Client-Side Web Refactorings (CSWR) [Garrido13Pers] y Usability Smells Finder (USF) [Grigera17], las cuales ayudaron a comprender el concepto de bad smells, tanto la forma de cómo pueden ser detectados, reportados y además brindar una solución aplicada del lado del cliente en términos de *refactoring*.

Además, se mencionaron y citaron herramientas previamente desarrolladas en diferentes trabajos de investigación para la detección de usability y accessibility smells. Como lo son Kobold [Grigera18SR] y Kobold en su versión accesible [Durgam2020]. También se describió en detalle la herramienta desarrollada por Watanabe [Watanabe16] y sus metodologías utilizadas, las cuales sirvieron como punto de partida para el desarrollo de varios de los *finders* definidos en esta tesina.

Una de las principales ventajas de la *Herramienta de Detección Automática de Accessibility Smells* es que su código es público, así que puede ser extendido sin limitaciones para lograr detectar más casos de elementos web inaccesibles. Otra ventaja importante es que los *finders* definidos en la extensión web están desarrollados en JavaScript, el cual es el lenguaje programático más utilizado en el mundo del desarrollo de páginas web, su estructura es de conocimiento universal. Por lo tanto, la gran mayoría de desarrolladores web podrán extender la herramienta desarrollando nuevos finders de forma rápida y sencilla dependiendo la complejidad del *accessibility smell* a detectar.

En este trabajo además de proveer una herramienta extensible, el desarrollo se basó en una arquitectura de sistemas modernos donde en vez de tener toda la herramienta centralizada en una sola aplicación, esta está dividida en tres capas o aplicaciones diferentes. La principal ventaja de esto, es que se puede trabajar en una de las capas sin afectar las demás. Por ejemplo: se puede desarrollar un nuevo *finder* en la extensión web sin que los demás dejen de funcionar y de poder ver los reportes con la información generada en la aplicación de reportes.

Los algoritmos de los *finders* desarrollados en esta tesina atacan diferentes tipos de elementos web generadores de *accessibility smells*, por lo cual se debieron utilizar diferentes técnicas y tecnologías de desarrollo para dichos *finders*. Estas técnicas pueden ser de gran ayuda como punto de partida para la extensión de la herramienta de detección, ya que pueden ser reutilizadas para la definición de nuevos *finders*.

Se dedicaron capítulos específicos para describir como fueron desarrollados los *finders* para diferentes problemas de accesibilidad. En ellos se destacaron los diferentes métodos, técnicas y API's JavaScript utilizadas para lograr detectar los elementos web problemáticos. También se mencionaron varios sitios web actuales que poseen estos elementos problemáticos, para los cuales se ejecutó la herramienta sobre ellos para demostrar su efectividad acompañado de los reportes generados. Estos contienen diversa información útil sobre cada uno de los elementos generadores de *accessibility smells*. Esta información es de gran ayuda para que los desarrolladores puedan encontrar fácilmente para luego revisar estos elementos inaccesibles, y así poder decidir la mejor forma de adaptarlos para que puedan ser accesibles.

7.1.1 Medición de performance

Para verificar el tiempo de ejecución del código de la extensión web, se utilizaron dos funcionalidades nativas de los navegadores que son: **Date.now()** y **performance.now()**. Ambas se utilizan para calcular el tiempo de ejecución de código JavaScript. Se corrieron ambas funciones en el primer script JS que es ejecutado al comienzo de la extensión web, y luego se volvieron a ejecutar en el último script de la extensión, con los siguientes resultados:

Sitio Web	Ejecución #1 Date|Performance	Ejecución #2 Date|Performance	Ejecución #3 Date|Performance	Ejecución #4 Date|Performance	Promedio Date|Performance
Walmart.com	131 | 131	238 | 238	292 | 291	211 | 209	218 | 217.25
Buenosaires.gob.ar	23 | 26	37 | 38	37 | 37	38 | 38	33.75 | 34.75
Eldia.com	56 | 54	51 | 51	64 | 65	49 | 50	55 | 55
Arba.gov.ar	25 | 24	27 | 28	28 | 28	30 | 30	27.5 | 27.5

Tiempos expresados en ms.

Como primera conclusión, se puede observar que ambas funciones arrojaron resultados similares. Por otro lado, el promedio de ejecución del código JS de la extensión web es muy aceptable, en ninguna de las pruebas llegó a superar el segundo de duración. Y como era de esperarse, en los sitios web con más cantidad de elementos, el tiempo de ejecución se incrementa.

7.2 Trabajos Futuros

A lo largo del desarrollo de esta herramienta han surgido diferentes ideas que pueden ser implementadas a futuro para continuar con la mejora y ampliación de funcionalidades de la herramienta. Como trabajos futuros se plantean los siguientes puntos:

- **Ampliación del scope de la herramienta para poder aplicar refactorings**: actualmente la herramienta cumple las funciones de detectar y reportar *accessibility smells*. Sería una gran mejora poder incluir la aplicación de *refactorings* de código a los elementos detectados y reportados por la herramienta cuando sea posible, y así lograr aplicar una solución a estos problemas.
- **Mejora de la interfaz de la aplicación de reportes**:
    - **Agregar roles de usuario**: para lograr una mejor protección de los datos que se muestran en los reportes, y así lograr restringir el acceso a los reportes de una determinada página web a sus respectivos desarrolladores y dueños. Además de que los dueños tendrían más privilegios y acceso a acciones que los usuarios comunes y desarrolladores no tendrían, por ejemplo: limpiar los datos de una determinada página web y así refrescar sus reportes. O cuáles *finders* van a estar detectando y reportando elementos problemáticos en un sitio web determinado.
    - **Mejorar la interfaz de los reportes**: actualmente la interfaz es bastante simple, pero al estar desarrollada en una tecnología tan nueva como lo es Angular existen infinidad de formas de presentar la información. Sería bueno implementar una nueva interfaz que posea más utilidades para los desarrolladores y dueños.
- **Integrar la herramienta de detección en Kobold**: como se explicó en el capítulo de trabajos relacionados, Kobold es una herramienta automática de detección para problemas de usabilidad muy completo. Realiza las tareas de búsqueda, detección y reporte de *bad smells*. Por lo tanto, como trabajo a futuro se podría plantear la integración de la herramienta de detección de esta tesina con Kobold. Esto lograría aprovechar todo el trabajo realizado en los procesos de detección y reporte de Kobold, y así poder ampliar su catálogo de *bad smells* de accesibilidad.
- **Poder importar la herramienta de detección como un archivo script JavaScript**: como sabemos, actualmente la detección de la herramienta está incluida en una extensión web. Esto es para que el código JavaScript de los *finders* de búsqueda se ejecuten sobre el sitio web donde está ejecutándose la extensión. Por lo tanto, se podría migrar este código JS a un archivo script de tipo .js e importarlo desde el sitio web

- **Escalar la herramienta a un ambiente de producción**: para esta tesina se probó el funcionamiento de la herramienta en un ambiente de desarrollo local. En el anexo del capítulo 8 se encuentran las instrucciones para que cualquier persona que quiera instalar la herramienta de forma local pueda hacerlo sin problemas. Por lo tanto, la herramienta de detección solo va a reportar lo que suceda en el browser de la persona que tenga instalada la extensión web y solo ella podrá visualizar esos reportes.

  Sin embargo, la arquitectura que se utilizó para el desarrollo de la herramienta, permite que se pueda ampliar el alcance de la misma sin tener que realizar grandes cambios. Resumiendo, como trabajo futuro se propone promover la API REST, base de datos y aplicación de reportes de esta herramienta a un servidor de aplicaciones en red (base de datos y web). Esto permitirá que varios usuarios reporten a una misma base de datos, y así poder generar reportes de accesibilidad con datos generados por múltiples usuarios y no solamente por una misma persona.

- **Mejora de los finders de la herramienta:** durante el desarrollo de los finders de la herramienta para esta tesina, se fueron mejorando y ajustando a medida que se iban probando en diferentes sitios web. Esto logró que la efectividad sea alta, como quedó demostrado en los resultados de los reportes, pero aun así quedaron algunos ajustes por realizar.

  - **Finder para la detección de elementos con handlers JS:** con el paso del tiempo JavaScript sigue siendo más popular para el desarrollo de aplicaciones web, por lo tanto, se siguen incorporando nuevas funcionalidades al lenguaje, en el caso de este *finder* quedaron algunos elementos que no pudieron ser detectados. Sin embargo, pueden ser incorporados a la detección si en el algoritmo del *finder* se incluye en la búsqueda elementos con handlers de tipo **mousedown**, **mouseup**, **touchend** y **touchstart** entre otros. Se propone como trabajo futuro tratar de incluir estos eventos para lograr ampliar la detección del finder.
  - **Finder para la detección de mensajes dinámicos en formularios inaccesibles:** durante las pruebas de este *finder* en diferentes formularios web, los resultados dieron que la detección fue muy satisfactoria logrando detectar la gran mayoría de mensajes de errores. Aun así, se observó que algunos formularios despliegan estos errores en secciones (s) o pop-ups que se encuentran fuera del formulario. Esto hace que estos mensajes no sean detectados por el *finder*, ya que el algoritmo sólo busca por dentro del formulario. Se propone como trabajo futuro tratar de ampliar la búsqueda de mensajes de error por fuera de la declaración del formulario.

- **Finder para la detección de elementos web flotantes:** durante las pruebas de este finder, vimos que hubo al menos un caso de un pop-up no fue detectado debido a que los cambios de estilos CSS del elemento no fueron capturados mediante el uso de *Mutation Observers*. Por lo tanto, como trabajo futuro se plantea incluir otros métodos para la detección de cambios CSS en los elementos de la página web. Además, se debe realizar una búsqueda interna de los elementos hijos del elemento flotante, ya que, si estos poseen atributos para lograr accesibilidad, pueden lograr que el elemento flotante termine siendo accesible.

CAPÍTULO 8

BIBLIOGRAFÍA Y ANEXO

8.1 REFERENCIAS BIBLIOGRÁFICAS

[AccWebForm] Conceptos generales sobre la accesibilidad de formularios. http://accesibilidadweb.dlsi.ua.es/?menu=accesibilidad-formularios-general. Accedido el 6/4/2021.

[AccPrinc] Principios de accesibilidad.

https://www.w3.org/WAI/fundamentals/accessibility-principles/es. Accedido el 6/4/2021.

[Accessuse] Access & Use site, basado en los resultados del proyecto COMPARE. Fundado por la Unión Europea vía el programa Erasmus+.

[AChecker] Achecker. Web Accessibility Checker. https://achecker.ca/.

[Antonelli2018] Antonelli, H. L., Igawa, R. A., Fortes, R. P. D. M., Rizo, E. H., & Watanabe, W. M. (2018). Drop-down menu widget identification using HTML structure changes classification. ACM Transactions on Accessible Computing (TACCESS), 11(2), 1-23.

[Ball 2013] BALL, D. 2013. I thought title text improved accessibility, I was wrong. http://silktide.com/ithought-title-text-improved-accessibility-i-was-wrong/.

[Buschmann08] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (2008). Pattern-Oriented Software Architecture: A System of Patterns, Volume 1 (Vol. 1). John wiley & sons.

[Caballero2012] Caballero, M. S. (2012). Accesibilidad y usabilidad en la Sociedad Digital. Antena de telecomunicación, (185), 12-17.

[CssTricks] Random Interesting Facts on HTML/SVG usage.

https://css-tricks.com/random-interesting-facts-htmlsvg-usage/#article-header-id-11. Accedido el 23/6/2021.

[ChromeDevTools] Documentación sobre herramientas para el desarrollo web provistas por Google Chrome. https://developer.chrome.com/docs/devtools. Accedido el 29/6/2021.

[ChromeExtensions] Google Chrome docs sobre la Extension API de Chrome. https://developer.chrome.com/docs/extensions. Accedido el 8/5/2021.

[Digital-Discrimination] The high cost of digital discrimination: why companies should care about web accessibility. Thu 31 Dec 2015. Artículo obtenido de www.theguardian.com. Accedido el 18/7/2021.

[DOMWiki] Document Object Model.

https://en.wikipedia.org/wiki/Document_Object_Model. Accedido el 5/4/2021.

[Durgam2020] Durgam, F. (2020). Detección de problemas de accesibilidad en la utilización de lectores de pantalla en aplicaciones web (Doctoral dissertation, Universidad Nacional de La Plata).

[FormNotif] Forms's Notifications. https://www.w3.org/WAI/tutorials/forms/notifications/. Accedido el 6/4/2021.

[Fowler99] Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston, Massachusetts, USA.

[Gamma+95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley. 1995

[Garrido11] Garrido, A., Rossi, G., Distante, D.: Refactoring for usability in web applications. IEEE Softw. 28(3), 60–67 (2011).

[Garrido13Imp] Garrido, A., Rossi, G., Medina-Medina, N., Grigera, J., Firmenich, S. Improving Accessibility of Web interfaces: refactoring to the rescue. Univ. Access in the Information Society 2013.

[Garrido13Pers] Garrido, A.; Firmenich, S.; Rossi, G.; Grigera, J.; Medina, N.; Harari, I. Personalized Web Accessibility using Client-Side Refactoring. IEEE Internet Comput. To appear 2013.

[Grigera17] Grigera, J; Garrido, A.; Rivero, J. M.; Rossi, G. Automatic detection of usability smells in web applications. International Journal of Human-Computer Studies. 97. 10.1016/j.ijhcs.2016.09.009. 2017.

[Grigera18SR] Grigera, J. (2018). Self-Refactoring: mejoras automáticas de usabilidad para aplicaciones web (Doctoral dissertation, Universidad Nacional de La Plata).

[ISO11] ISO, I (2011) ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements.

[María and Paz 2012] MARÍA, M. AND PAZ, L. 2012. "Accesibilidad y Usabilidad: los requisitos para la inclusión digital.". In VII Jornadas de Sociología de la UNLP 5 al 7 de diciembre de 2012 La Plata, Argentina.

[Medina10] Medina-Medina, N., Rossi, G., Garrido, A., & Grigera, J. (2010). Refactoring for Accessibility in Web Applications. In Proceedings of the XI Congreso Internacional de Interacción Persona-Ordenador (INTERACCIÓN 2010), Valencia, Spain (pp. 427-430).

[MozillaWebAPIs] Web APIs disponibles para el desarrollo de sitios y aplicaciones webs. https://developer.mozilla.org/en-US/docs/Web/API. Accedido el 5/4/2021.

[Nielsen06] Nielsen, J., & Loranger, H. (2006). Prioritizing web usability. Pearson Education.

[NVDA] NVDA, NonVisual Desktop Access. https://www.nvaccess.org/.

[LEVELACC] Level Access. Digital Accessibility. https://www.levelaccess.com

[OAWEC] Observatorio de Accesibilidad Web. Ecuador.

https://www.consejodiscapacidades.gob.ec/observatorio-de-accesibilidad-web/.

[ObsVsPub-Sub] Observer vs Pub-Sub Pattern.

https://betterprogramming.pub/observer-vs-pub-sub-pattern-50d3b27f838c. Accedido el 4/4/2021.

[Peter Krantz 2005] PETER KRANTZ. 2005. Browsing habits of screen reader users. http://www.standardsschmandards.com/2005/browsing-habits/.

[PromVsObs] JavaScript Theory: Promise vs Observable.

https://medium.com/javascript-everyday/javascript-theory-promise-vs-observable-d3087bc1239a/. Accedido el 4/4/2021.

[TAW] Taw. Test de accesibilidad web. https://www.tawdis.net/.

[UXM] UX Movement. https://uxmovement.com/

[W3C] World Wide Web Consortium. https://www.w3.org/.

[W3C05] W3C. 2005 Web Accessibility Initiative (WAI). Accessibility Introduction. https://www.w3.org/WAI/fundamentals/accessibility-intro/

[WAIARIA] WAI ARIA, the Accessible Rich Inter net Applications Suite.

https://www.w3.org/WAI/standards guidelines/aria/.

[WCAG] WCAG 2.1 2018. Web Content Accessibility Guidelines.

https://www.w3.org/TR/WCAG21/.

[WebAIM] Web Accessibility in mind, Creating Accessible Forms.

https://webaim.org/techniques/forms/. Accedido el 7/4/2021.

[Watanabe16] Watanabe, W. M., & de Mattos Fortes, R. P. (2016, April). Automatic identification of drop-down menu widgets using mutation observers and visibility changes. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (pp. 766-771).

8.2 ANEXO

Cómo ejecutar la herramienta

Este anexo tiene la función de dar una explicación de cómo ejecutar la herramienta de detección en un ambiente local. Para esto se definieron los pasos para poder ejecutar localmente todos los componentes de la herramienta.

Dividiremos esta sección por componente: *Extensión Web*, *API REST - Base de Datos*, y la *Aplicación de Reportes*.

Se necesita tener la extensión web, API REST y la Base de datos ejecutándose para lograr almacenar la información de los smells detectados por la extensión web.

Para lograr ver estos datos de una forma legible, se debe ejecutar la aplicación de reportes.

Tener pre-instalado:

- NODE JS: versión > 12.0.0, usada para este trabajo v12.14.0
- Angular CLI: versión 10.2.1
- MONGODB
- Web Browser: Mozilla Firefox o Google Chrome.

8.2.1 Extensión Web

Para instalar la extensión web en nuestro navegador, primero necesitamos descargar su código fuente del repositorio.

Extensión Web – Link del repositorio público

https://github.com/tole22/tesinaLS/tree/master/Accessibility-web-extension

Una vez descargado, podremos comenzar con la instalación en el web browser. Para este trabajo se pudo determinar que la extensión funciona correctamente tanto para **Mozilla Firefox** como para **Google Chrome**.

La forma de instalación en estos web browsers es la siguiente:

Para Mozilla Firefox

Debemos entrar a la siguiente url:

about:debugging#/runtime/this-firefox

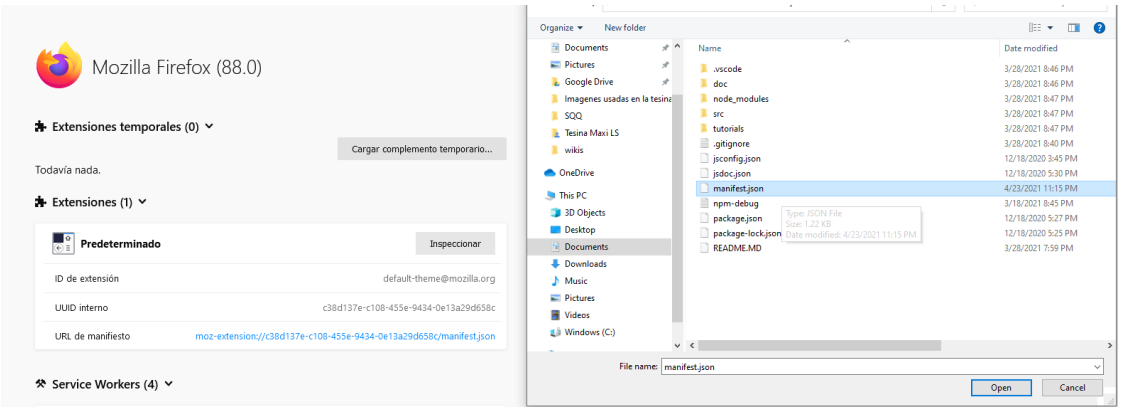


Figura 8.1: Cargar extensiones temporales en Firefox.

Como vemos en la Figura 8.1 podremos instalar extensiones temporales. Seleccionamos “Cargar complemento temporario” y seleccionamos el archivo *manifest.json* de nuestra extensión web.

Esto instalará la extensión web “*Accessibility-BadSmells-Finder*”, Figura 8.2, la cual ya estará reportando una vez que ingresemos a un sitio web.

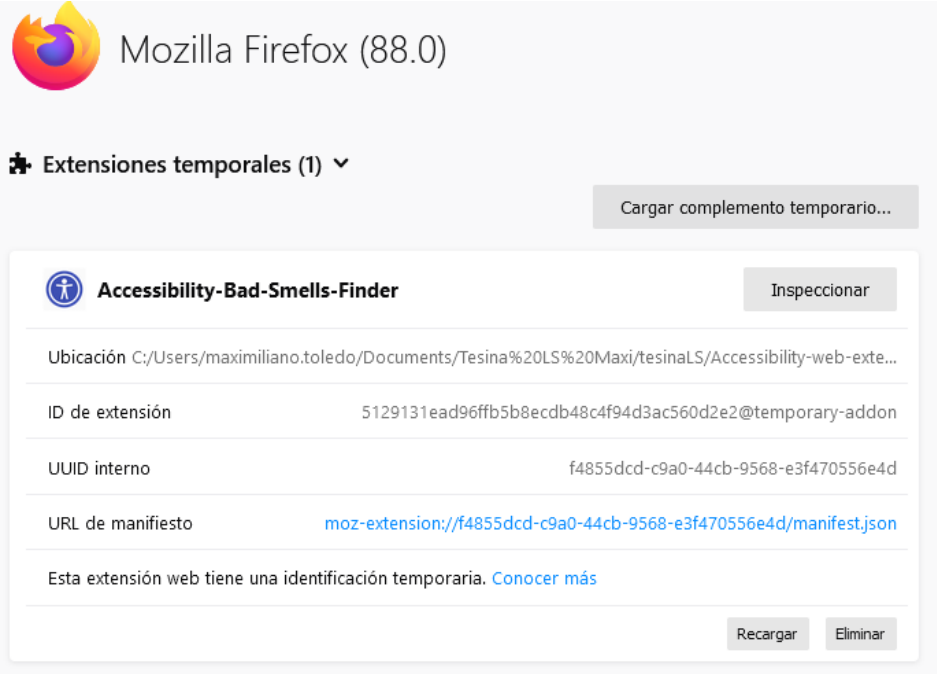


Figura 8.2: Web extension instalada.

En Google Chrome

Debemos entrar a la siguiente url dentro del browser:

chrome://extensions/

Luego, como vemos en la Figura 8.3, debemos seleccionar “Load unpacked” (paso #1). Nos pedirá seleccionar la carpeta donde se encuentra el código fuente de la extensión web (paso #2).

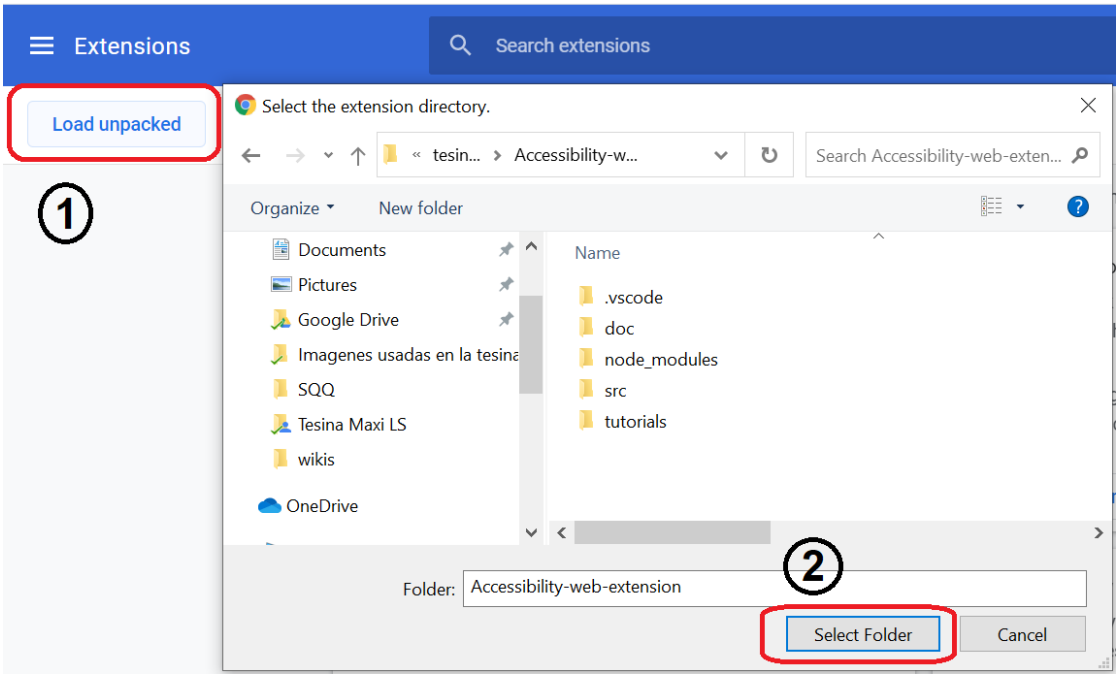


Figura 8.3: Cargar extensiones temporales en Chrome.

Luego de esto, la instalación de la extensión ya estará completada, Figura 8.4.

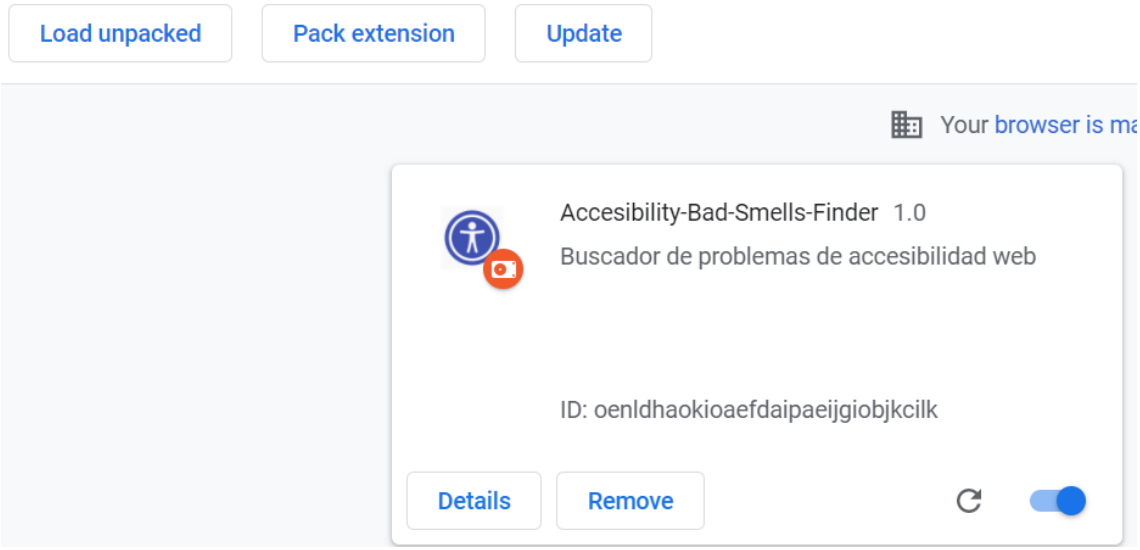


Figura 8.4: Extensión web instalada correctamente en Chrome.

Para validar si la extensión está funcionando, se puede comprobar en la consola del navegador los logs que la misma va logeando: ejemplo en la Figura 8.5.

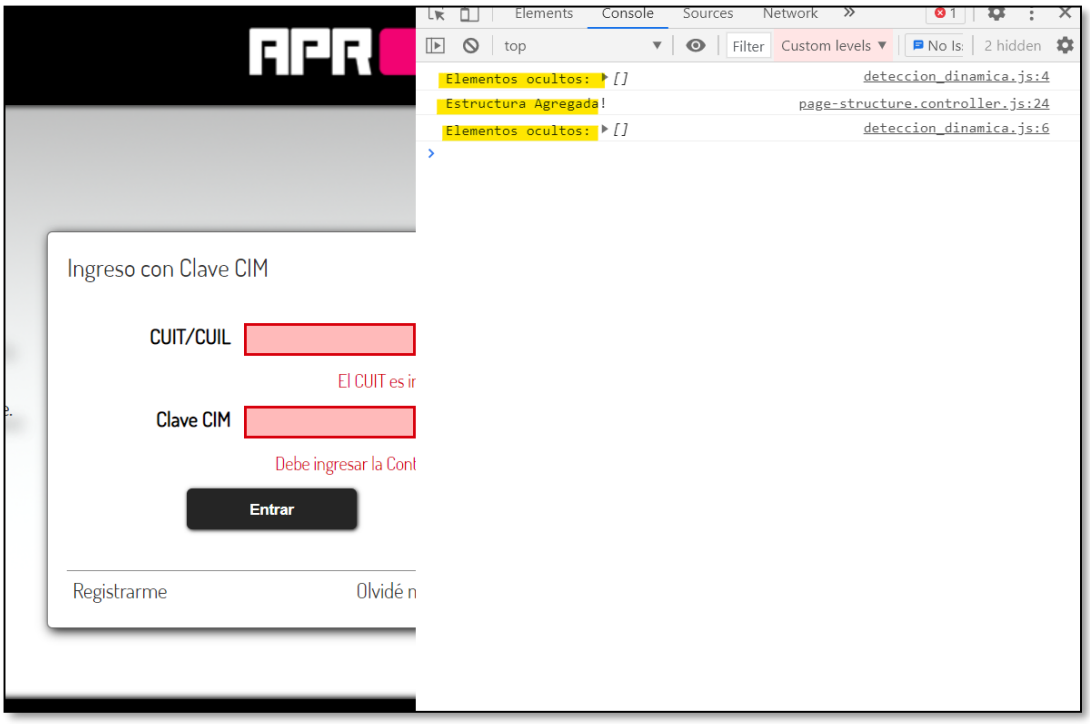


Figura 8.5: Ejemplo de los logs que va generando la extensión web del a herramienta de detección.

Nota: Por defecto la extensión web está configurada para ejecutarse en cualquier sitio web, pero esto se puede restringir a páginas web especificas en la sección "*matches*" del archivo *manifest.json*.

8.2.2 API REST y Base de datos

Base de datos

Solo se debe instalar MongoDB, y luego en una consola del sistema ejecutar el comando ***mongod***. Como observamos en la Figura 8.6.

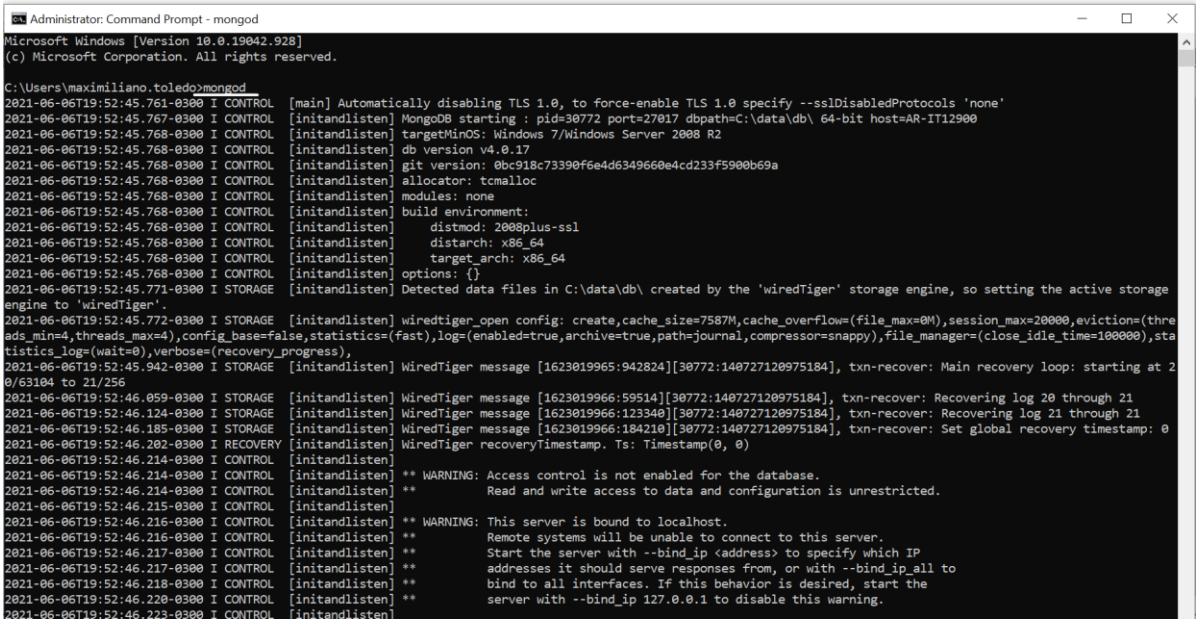


Figura 8.6: Ejecutando la base de datos localmente utilizando el comando *mongod*.

API REST

Para tener en funcionamiento la API REST, primero necesitamos descargar su código fuente del repositorio.

API REST – Link del repositorio público

https://github.com/tole22/tesinaLS/tree/master/rest-api-events-web-extension-Tesina

Una vez descargado, podremos comenzar con la instalación de las dependencias de Node que utiliza esta API. Para esto necesitamos:

1. Abrir una consola de sistema en la carpeta del código fuente de la API.
2. En la consola ejecutar el comando ***npm install***
3. Una vez que termine la instalación, podremos ejecutar la API REST con el comando ***npm run dev***

Con esto ya estaríamos permitiendo que la extensión web almacene eventos, estructuras, smells y demás información en la base de datos a través de la API REST.

8.2.3 Aplicación de Reportes

Para tener el entorno completo de la herramienta en ejecución, debemos tener la aplicación de reportes instalada y ejecutándose para lograr visualizar el resultado final expresado en un reporte.

Los pasos son tan simples como con la API REST:

1. Descargamos su código fuente:

APP de reportes – Link del repositorio público

https://github.com/tole22/tesinaLS/tree/master/reportes-app-Tesina/app-reportes-tesina

2. Abrir una consola de sistema en la carpeta donde descargamos su código fuente.
3. En la consola ejecutar el comando ***npm install*** para descargar todas sus dependencias.
4. Una vez que termine la instalación, podremos ejecutar la aplicación de reportes con el comando ***ng serve***

Para visualizar la aplicación debemos ingresar a `*http://localhost:4200/*` en cualquier navegador web, como se observa en la Figura 8.7.

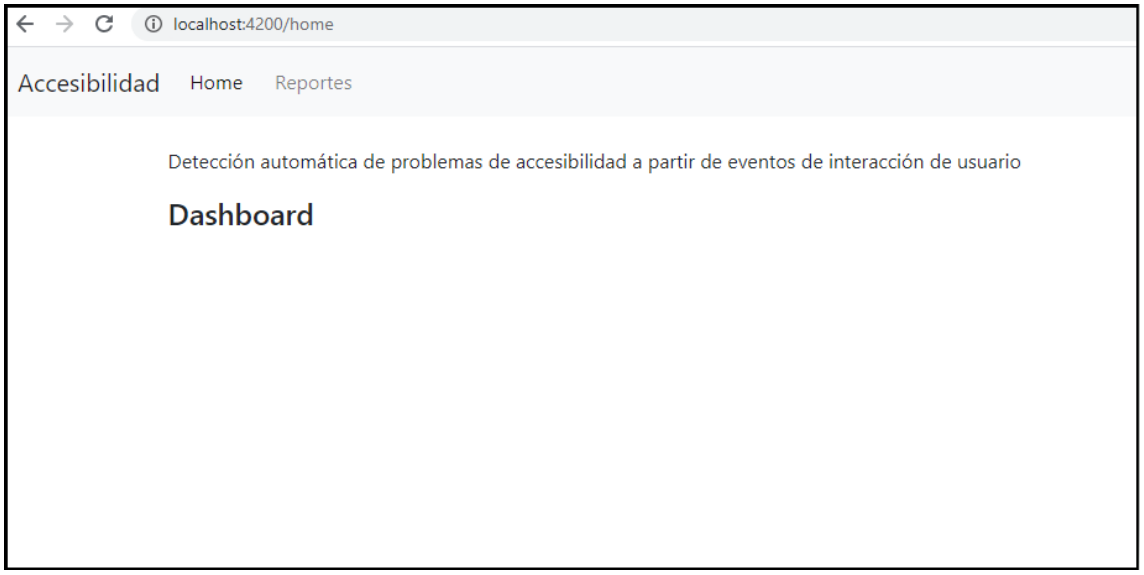


Figura 8.7: Ingresando a la Aplicación de reportes a través del navegador web.